PC-8001B N-Basic
Reference Manual

NEC

PC-8102B
PTS-069

PC-8001B N-Basic Reference Manual

NEC

IMPORTANT NOTICE

(1) All rights reserved. This manual is protected by copyright. No part of this manual may be reproduced in any form whatsoever without the written permission of the copyright owner.
(2) The policy of NEC being that of continuous product improvement, the contents of this manual are subject to change, from time to time, without notice.
(3) All efforts have been made to ensure that the contents of this manual are correct; however, should any errors be detected, NEC would greatly appreciate being informed.
(4) NEC can assume no responsibility for errors in this manual or their consequences.

CONTENTS

CONTENTS (continued)

CONTENTS (continued)

CONTENTS (continued)

CONTENTS (continued)

CONTENTS (continued)

FIGURES

TABLES

CHAPTER 1

GENERAL INFORMATION

All the N-Basic program statements described in this manual are in alphabetical order for easy reference. Each command is followed by a brief description, its syntax, and examples. All syntax descriptions follow the standards listed in Table 1-1.

Table 1-1 Syntax Description Standards

STANDARDS	EXAMPLES
Lowercase lettering indicates a mandatory user-defined field.	SAVE "filename"
Items enclosed in brackets are optional.	BEEP [switch]
A list of items enclosed in braces implies one of the items must be used.	FRE ({0 / A$})
All other punctuation is entered as shown.	CSAVE "PROB1"

MODES OF OPERATION

The PC-8000 can be used in three modes: direct, indirect, and terminal. The direct mode is entered when the system is turned on. In direct mode, statements and commands are executed as entered. The results are displayed immediately, but the instruction is lost. The direct mode is useful for debugging and using the PC-8000 as a calculator for quick computations that do not require a complete program.

You use the indirect mode to create programs. Program statements are preceded by line numbers that are stored in memory and later executed by a RUN command.

Terminal mode, entered by issuing a TERM command, enables the PC-8000 to serve as a terminal for another computer. Typing a Control-B (hold the CNTR key, press the B key) returns control to direct mode.

N-Basic consists of commands and statements. Statements can be used interchangeably in direct and indirect modes. Commands can only be used in direct mode. This manual denotes commands with an asterisk after the name.

LINE FORMAT

A program line always begins with a line number and ends with a carriage return. A program line can contain a maximum of 255 characters and follows this syntax:

nnnn BASIC statement [:BASIC statement . . .]

More than one statement to a line can be specified by separating each program statement with a colon.

CHARACTER SET

The PC-8000 character set is composed of alphabetic, Greek, numeric, and other special characters. For a complete list of available characters, see the Character Code Chart in Appendix D.

CONTROL CHARACTERS

N-Basic comes complete with a set of special control characters that perform the special functions listed in Table 1-2. Execute control characters by holding down the CNTR key and pressing the letter key.

Table 1-2 Control Character Functions

CONTROL CHARACTER	FUNCTION
Control-B	Moves the cursor to the head of the preceding item.
Control-C	Terminates input operations and returns control to direct mode.
Control-E	Deletes all characters following the cursor in the current line.
Control-G	Sounds the PC-8000 buzzer.

Table 1-2 Control Character Functions (cont'd)

CONTROL CHARACTER	FUNCTION
Control-H	Moves the cursor one position to the left and clears that position (destructive backspace).
Control-J	Shifts all characters from the cursor to the end of the line, to the head of the next line.
Control-I	Tab space every eight columns.
Control-K	Moves the cursor to its home position, which is the upper left corner of the screen.
Control-L	Clears the screen.
Control-N	Moves the cursor to the head of the next item displayed on the screen.
Control-R	Shifts all characters right of the cursor one space and leaves the cursor position blank.

In addition to the control characters, N-Basic uses the control keys STOP and ESC (escape). STOP halts program execution and returns the system to direct mode. ESC halts program execution or listing and waits for input. Any character typed during this wait state continues program execution.

CONSTANTS

The two types of constants used in N-Basic are string and numeric.

String Constants

A string constant is a sequence of up to 255 alphanumeric characters enclosed in quotation marks.

"HELLO"
"$25,000.00"
"Number of Employees"

Numeric Constants

Numeric constants can be positive or negative and can be one of the types listed in Table 1-3.

Table 1-3 Numeric Constants

TYPE	DESCRIPTION
Integer constants	Whole numbers between −32768 and +32767. Example: A%=5001
Fixed point	Positive or negative real numbers. Example: A=35.54
Floating point	Positive or negative numbers represented in exponential form. A floating-point constant consists of a mantissa followed by the letter E and the exponent. The exponent must be in the range of −38 to +38. Double precision floating-point constants use the letter D. Both letters D and E must be capital letters. Example: X#=2359E+6
Hex	Hexadecimal numbers are prefixed by &H. Hex numbers entered in this format will be output in decimal. Example: 10 X=&H76 20 PRINT X run 118
Octal	Octal numbers are prefixed by &0 or &. Octal numbers are also displayed in decimal. Example: 10 X=&0347 20 PRINT X run 231

Single and Double Precision

Numeric constants can be single or double precision. Single precision constants consist of any numeric value that has seven or fewer digits, exponential form, or a trailing exclamation point (!). Six of the seven significant digits are displayed.

Double precision constants have eight or more digits, exponential form using D, or a trailing number sign (#). Double precision constants have 17 significant digits, 16 of which are displayed.

Examples

a) Single Precision

46.8
−1.09E−06
3489.0
22.5!

b) Double Precision

34569811
−1.09432D−06
3489.0#
7654321.1234

VARIABLES AND DECLARATION CHARACTERS

N-Basic variable names can be any length; however, only the first two characters are significant. The first character of a name must be an alphabetic character; the remaining characters can be alphanumeric.

A variable name cannot be a reserve word, begin a reserve word, or contain a reserve word. For example, BFOR is illegal because it contains the reserve word FOR. Reserve words include all N-Basic commands, statements, and functions.

Variables can represent numeric or string values. String variable names are written with a dollar sign ($) as the last character (A$="Name"). The dollar sign is a variable type declaration character; it declares that the variable represents a string. You use any one of the four declaration characters to declare the following variable types.

CHARACTER	TYPE
$	String variable
%	Integer variable
!	Single precision variable
#	Double precision variable

If you omit a declaration character, the variable is assumed to represent a single precision value.

Examples

PI#	Declares a double precision value.
MINIMUM!	Declares a single precision value.
LIMIT%	Declares an integer value.
N$	Declares a string value.
ABC	Represents a single precision value.

Variables can also be declared as arrays by subscripting. For example, A(10) refers to a one-dimensional array of eleven elements, any elements of which can be referenced by A(0) through A(10). A(5,5) declares a two-dimensional array that contains 36 elements in 6 rows and 6 columns. Elements are referenced by A(0,0) to A(5,5).

TYPE CONVERSION

When necessary, N-Basic converts a numeric constant from one type to another. If you attempt to convert a string variable to numeric or numeric to string, a "Type mismatch" error occurs.

Observe the following rules when converting numeric constants.

a. If you set constants of different types, the constant is stored as the type declared in the variable name.

   Example

```
10 A%=23.42
20 PRINT A%
run
 23
```

b. During expression evaluation, all of the operands in an arithmetic or relational operation are converted, and their results returned, to the same degree of precision of the most precise operand.

Example

```
10D#=6#/7                    10 D=6#/7
20 PRINT D#                  20 PRINT D
run                          run
.8571428571428571            .857143
```

c. Logical operators convert their operands to integers and return an integer result. Operands must be in the range −32768 to 32767 or an "Overflow" error occurs.

Example

```
10 PRINT 8.123 OR 24
run
24
```

d. When a floating-point value is converted to an integer, the fractional portion is truncated.

Example

```
10 C%=55.88
20 PRINT C%
run
55
```

e. If a double precision variable is assigned to a single precision value, only the first seven digits, rounded down, of the converted number are valid because only seven digits of accuracy are supplied for a single precision value. The absolute value of the difference between the printed double precision number and the original single precision value will be less than 6.3E-8.

Example

```
10 A=2.04
20 B#=A
30 PRINT A;B#
run
2.04   2.039999961853027
```

EXPRESSIONS AND OPERATORS

An expression is a string or numeric constant that when combined produce a single value. Operators used in performing these mathematical or logical operations are divided into four categories: arithmetic, relation, logical, and functional.

Arithmetic Operators

Arithmetic operators, in the order of their precedence, are listed in Table 1-4.

Table 1-4 Arithmetic Operators

OPERATOR	OPERATION	SAMPLE EXPRESSION
^	Exponentiation	2^4
–	Negation	–2
*, /	Multiplication and floating-point division	2*4 4/2.5
\	Integer division	4\2
MOD	Modulus integer division	4 MOD 2
+, –	Addition and subtraction	4+2, 4–2

Use parentheses to change the order of operations. Operations within parentheses are performed first. Inside the parentheses, the normal order of operation is maintained.

Examples

a) Algebraic Expression	b) Basic Expression
$X+2Y$	X+2*Y
$X-\frac{Y}{X}$	X–Y/X
$\frac{X+Y}{Z}$	(X+Y)/Z
X^2Y	(X^2)*Y
XYZ	X* Y*Z
X (–Y)	X* (–Y)

Integer Division and Modulus Arithmetic

Integer division is denoted by the \ on your keyboard. Operands and quotients are rounded to integers.

Examples

```
10\5=2
25.68\6.99=4
```

Modulus arithmetic, denoted by the operator MOD, returns the integer value of the remainder of integer division.

Examples

10 MOD 3=1	10 divided by 3 with a remainder of 1
25.68 MOD 6.99=1	25 divided by 6 with a remainder of 1

Relational Operators

Relational operators compare two values and make decisions regarding program flow. The result of the comparison is true (1) or false (0). Available operators to N-Basic are listed in Table 1-5.

Table 1-5 Available Operators

OPERATOR	RELATION TESTED	EXPRESSION
=	Equality	X=Y
< > or > <	Inequality	X< >Y
< >	Less than Greater than	X<Y X>Y
<= or =< >= or =>	Less than or equal to Greater than or equal to	X<=Y X>=Y

Examples

```
IF SIN (X)<0GOTO 100
IF I MOD J<>0 THEN K=K+1
```

NOTE

If arithmetic and relational operators are combined in one expression, the arithmetic is always performed first. For example, the expression X+Y< (T-1)/Z is true if the result of X+Y is less than the result of T-1 divided by Z.

Logical Operators

N-Basic provides logical operators for performing bit manipulation, Boolean operations, or tests on multiple relations. As with relational operators, the logical operator returns a true (1) or false (0) value. Logical operations are performed after arithmetic and relational operations. Examples showing the outcome of logical operations follow. The operators are listed in the order of their preference.

Example

NOT	X	NOT X	
	1	0	
	0	1	
AND	X	Y	X AND Y
	1	1	1
	1	0	0
	0	1	0
	0	0	0
OR	X	Y	X OR Y
	1	1	1
	1	0	1
	0	1	1
	0	0	0
XOR	X	Y	X XOR Y
	1	1	0
	1	0	1
	0	1	1
	0	0	0

IMP	X	Y	X IMP Y
	1	1	1
	1	0	0
	0	1	1
	0	0	1

EQV	X	Y	X EQV Y
	1	1	1
	1	0	0
	0	1	0
	0	0	1

One use of logical operators is to connect two or more relational operators to make decisions on the direction of program flow. For example:

IF D<200 AND F<4 THEN 80	Both conditions would have to be true to branch to 80.
IF I>10 OR K<0 THEN 50	Unless both relational conditions are false, program control branches to line 50.

Logical operators convert their operands to 16 bit, signed, twos complement integers in the range of −32768 to +32767. The given operation is performed on these integers in a bit-by-bit fashion; thus, it is possible to use logical operators to test bytes for a particular bit pattern. In this vane, you can use the AND operator to mask all but one of the bits of a status byte at a machine I/O port, or you can use the OR operator to merge two bytes to create a particular binary value.

Examples

63 AND 16=16

63 equals binary 111111, 16 equals binary 100000, so a bit by bit AND operation yields 100000.

10 OR 10=10

10 equals binary 1010, so 1010 OR 1010 = 1010.

STRING OPERATIONS

Strings can be compared using the same relational operators that are used with numbers. String comparisons are made by taking one character at a time and comparing the ASCII codes. If the ASCII codes are the same, the string is considered equal. If the codes differ, the lower code precedes the higher. A shorter string is considered smaller. Leading and trailing blanks are significant.

Examples

"AAN" > "AB"	"FILENAME" = "FILENAME"
"X&" > "X#"	"CL" > "CL"
"kg" > "KG"	"SMYTH" < "SMYTHE"
B$ < "9/12/80"	where B$ = "8/12/80"

Strings can also be concatenated using +. For example:

```
10 A$="file": B$="name"
20 PRINT A$+B$
30 PRINT "new" +A$+B$
run
filename
new filename
```

NOTE

Strings used in comparison expressions must be enclosed in quotation marks.

SCREEN EDITOR

Characters entered from the keyboard are first received by the screen editor of N-Basic, then displayed at the current position of the cursor. Input from the keyboard is not interpreted by the interpreter until you enter a carriage return. Then lines that begin with a proper line number are stored in memory as program lines. Input without line numbers is interpreted in direct mode and operations are performed immediately. Changes to the text on screen are not made in memory unless you enter a carriage return.

CURSOR MOVEMENT

The PC-8000 has six functions for moving the cursor. The two direction keys located on the small keyboard operate in uppercase and lowercase and move the cursor in the indicated directions.

In addition to the direction keys, you can use Control-N to move the cursor forward to the next item and Control-B to move the cursor back one item.

INSERTION AND DELETION

The INS key moves all text following the cursor one space forward. The cursor does not move, allowing an insert in the middle of text. The same movement can be accomplished using Control-R.

The DEL key deletes text immediately preceding the cursor and moves all text following the cursor back one space. You can also use Control-H for this movement.

ERROR MESSAGES

Error messages for the PC-8000 are displayed in one of two ways. For the direct mode, the syntax is "XX message"; for indirect mode, the syntax is "XX message in nnn", where XX is the error code and nnn is the line number where the error is detected. For a complete list of N-Basic error codes, see Appendix C.

CHAPTER 2

STATEMENTS

This chapter describes the statements and commands of N-Basic.

AUTO*

AUTO automatically generates line numbers after every carriage return and switches control from direct mode to indirect mode. The two values that can be specified determine where the line numbers will begin and the increment they will increase by. The default for both values is 10.

To terminate the AUTO statement, enter a Control-C or press the STOP key. Control then passes to direct mode.

Syntax

AUTO [start line ,increment]

Examples

AUTO	Generates line numbers 10, 20, 30, . . .
AUTO 20, 15	Generates line numbers 20, 35, 50, 65, . . .
AUTO 50,	Generates line numbers 50, 65, 80, . . .
	The comma following 50 sets the increment default to the last value specified. In this example, 15.

BEEP

BEEP briefly sounds the PC-8001 buzzer.

Syntax

BEEP [switch]

---

*Commands should only be used in direct mode. This manual denotes commands with an asterisk after the name. The asterisk is not meant to be included in the spelling of the command.

Examples

BEEP	Briefly sounds the buzzer.
BEEP 1	Turns buzzer on.
BEEP 2	Turns buzzer off.

Note

You can use PRINT CHR$(7) in place of BEEP to briefly sound the buzzer.

CLEAR

CLEAR sets all numeric variables to the null characters. Its optional variables reserve string space and set the highest memory location available for use by an N-Basic program.

Syntax

CLEAR [string space ,high memory]

Examples

CLEAR	Sets all numeric and string variables to null.
CLEAR 500,49152	Sets all numeric and string variables to null, reserves 500 bytes of memory for string space, and sets high memory to 49152.

CLOAD*

CLOAD loads a program stored on cassette tape into memory. When CLOAD is executed, the system searches the tape for the specified program. Load a program name exactly as it is saved.

The prompt, SKIP program, appears, displaying all the programs skipped in the search. The prompt, FOUND:program,

is displayed when the program is located. A blinking asterisk appears, indicating the program is being loaded into memory.

CLOAD? compares a program currently in memory with a program of the same name on tape. If they are the same, OK appears. If not, BAD appears. The programs are not altered by this command.

Syntax

CLOAD "file name"

Examples

CLOAD "TEST1"	The program TEST1 is searched for on tape and loaded into memory.
CLOAD? "TEST1"	Compares the program TEST1 on tape with the program currently in memory.

NOTE

If problems are encountered using the CLOAD command, see Appendix F.

CLOSE

CLOSE ends access to (closes) all opened files and writes the remaining data in the output buffer to disk. You can open and close a file several times in the course of an N-Basic program.

Syntax

CLOSE [file number,file number]

Examples

CLOSE	Closes all opened files.
CLOSE 1,2	Closes file numbers 1 and 2.
CLOSE 2	Closes file number 2.

COLOR

COLOR assigns a color or a screen attribute to a display. It has three options.

a. In color mode, you can specify a color.

b. In black-and-white mode, you can specify a screen attribute.

c. In either mode, you can specify a graphic switch.

Code determines the colors or attributes described in Table 2-1.

Table 2-1 COLOR Options

COLOR CODE	COLOR (COLOR MODE)	ATTRIBUTES (BLACK AND WHITE MODE)
0	Black	Normal
1	Blue	Secret
2	Red	Blink
3	Magenta	Secret
4	Green	Reverse Field
5	Cyan	Reverse Field-Secret
6	Yellow	Reverse Field-Blink
7	White	Reverse Field-Secret

Optionally, you can specify the null character code, which is the ASCII character displayed when the screen is cleared by a PRINT CHR$(12) statement or the HOME CLR key. You normally specify 0 to clear the screen.

The graphic switch option chooses graphic or character mode. A 1 turns on the graphic switch so you can use the graphic capabilities of the PC-8000. A 0 turns on the character mode.

Syntax

COLOR code [,null character code ,graphic switch]

Examples

COLOR 6,0,1	The color is yellow, the null character code is 0, and the graphic switch is turned on.
COLOR 2	In color mode, the screen is set to red. In black and white mode, the screen attribute is blink.
COLOR 4,,0	The color is green and the character mode is switched on.

CONSOLE

CONSOLE formats the screen and determines the color mode.

Syntax

CONSOLE scroll line [,scroll length ,key switch ,color switch]

The default values for these options are listed in Table 2-2.

Table 2-2 CONSOLE Options Default Values

OPTION	DEFAULT
Scroll line	None
Scroll length	From scroll line to bottom of screen
Key switch	1 (function key display on)
Color switch	0 (black and white)

Examples

CONSOLE 2	Scrolls from the bottom of the screen to line 2, the function key display is on, the mode is black and white.
CONSOLE 2,10	Scrolls lines from 11 to 2, the function key display is on, the mode is black and white.

CONSOLE 2,10,0,1	Scrolls lines from 11 to 2, the function key display is off, the mode is color.

CONT*

CONT resumes execution of a program after you press the STOP key or execute a STOP or END statement. Execution resumes at the point the break occurred.

Syntax

CONT

Examples

a)
```
10 FOR I=1 TO 100
20 PRINT I
30 STOP
40 NEXT I
ok
run
1
Break in 30
ok
CONT
2
BREAK IN 30
CONT
3
.
.
.
```

b)
```
10 FOR I=1 TO 100
20 PRINT I
30 NEXT I
ok
run
1
2
3
4
^C STOP key pressed here
CONT
5
6
7
.
.
.
```

CSAVE

CSAVE stores the file currently in memory on cassette tape. You can use it in direct mode or in a program statement.

Syntax

CSAVE "file name"

Example

```
CSAVE "PROB1"
```

Stores the current file "PROB1" on cassette tape.

DATA

DATA supplies data items to a READ statement. The READ statements access the DATA statements in order by line number. Thus, the items contained in the DATA statements can be considered one continuous list. A DATA statement can contain as many constants as fit a line and can be placed anywhere in a program.

Syntax

```
DATA constant,constant . . .
```

Example

```
10 REM PROGRAM TO READ LIST
20 REM OF VARIABLES AND PRINT
30 REM THEM OUT.
40 FOR I=1 TO 3
50 READ A,B,C
60 PRINT A;B;C
70 NEXT I
80 DATA 1,2,3
90 DATA 4,5,6
100 DATA 7,8,9
110 END
RUN
 1  2  3
 4  5  6
 7  8  9
```

DEF

DEF declares a variable or a range of variables as integer (%), single precision (!), double precision (#), or string variable ($) type. If a type declaration character is encountered in a program statement, it takes precedence. The system default is single precision.

Syntax

DEF { INT / SNG / DBL / STR } { variable, variable, ... / variable — variable }

Examples

a)
```
10 DEFDBL D
20 D=3/5
30 PRINT D
run
6000000238418579
```

b)
```
10 DEFDBL D
20 D!=3/5
30 PRINT D!
run
.6
```

c)
```
10 DEFSTR A-D
20 A="HI":B="HOW"
30 C="ARE":D="YOU"
40 PRINT A,B,C,D
run
HI            HOW
ARE           YOU
OK
```

d)
```
10 DEFDBL L
20 LOW=3/5
30 PRINT LOW
run
6000000238418579
```

By beginning with the same letter declared in line 10, low is assumed a double precision variable.

DEFFN

You use DEFFN to define functions not intrinsic to the PC-8000. The function name must be preceded by FN, and the variables in its argument list must be separated by commas. You can define functions as numeric or string, but their argument type must match. Execute a DEFFN statement to define a function before it is called.

Syntax

DEFFN function name (argument list)=function definition

Examples

a) Numeric Function Definition

```
10 DEFFNB (X,Y)=X/Y*100
20 I=20:J=5
```

```
30 T=FNB (I,J)
40 PRINT T
50 END
run
400
```

b) String Function Definition

```
10 DEFFNB$ (X$,Y$)=X$+Y$
20 I$="ABC":J$="DEF"
30 T$=FNB$ (I$,J$)
40 PRINT T$
50 END
run
ABCDEF
```

c) Type Mismatch Error

```
10 DEFFNB$ (X,Y)=X$+Y$
20 I$="ABC":J$="DEF"
30 T$=FNB$ (I$,J$)
40 PRINT T$
50 END
run
Type Mismatch in 30
```

The type mismatch error is caused because the function arguments in line 10 (X,Y) are numeric data types and the function call arguments, line 30, are strings (I$,J$).

d) Undefined Function Error

```
10 I$="ABC":J$="DEF"
20 T$=FNB$ (I$,J$)
30 PRINT T$
40 DEFFNB$ (X,Y)=X$+Y$
50 END
run
Undefined User Function in 20
```

An attempt is made to call the function before it is defined.

DEFUSR

DEFUSR specifies the starting address of an assembly language subroutine. The subroutine is marked by a pointer, any

integer 0 to 9, which is the number of the USR routine whose address is being specified. Address is the starting address of the USR routine. See Appendix A, Assembly Language Subroutines.

Syntax

DEFUSR pointer=address

Example

```
200 DEFUSR0=24000
210 X=USR0 (Y 2/2.89)
```

DELETE*

DELETE erases specified program lines.

Syntax

DELETE line number [−to line number]

Examples

DELETE 40	Deletes line 40.
DELETE 40-70	Deletes lines 40 through 70.
DELETE-40	Deletes all lines up to and including 40.
DELETE	"ILLEGAL FUNCTION CALL". You must specify a line number. The same error occurs if you specify a line that does not exist.

Note

You can use a comma (,) in place of the dash (-).

DIM

DIM allocates storage for arrays and matrices. The system default for all subscripted variables used without the DIM statement is 10.

Although you can dimension a string, it may prove to be unnecessary. Strings are dynamically allocated up to 255 characters; in most cases, this amount of storage is enough. When it is not, use a CLEAR statement to increase the allocation of string space.

Syntax

DIM variable ([minimum value,] maximum value)

Examples

a)
```
10 DIM A(5)
20 FOR I=1 TO 5
30 A(I)= I
40 PRINT A;
50 NEXT I
run
1  2  3  4  5
```

b)
```
10 DIM A(5)
20 FOR I=1 TO 5
30 A(I)= I
40 PRINT A
50 NEXT I
run
1
2
3
4
5
```

c)
```
10 DIM A(5,5)
20 FOR I=1 TO 5
30 FOR J=1 TO 5
40 PRINT A(I,J);
60 NEXT J
60 PRINT
70 NEXT I
run
0  0  0  0  0
0  0  0  0  0
0  0  0  0  0
0  0  0  0  0
0  0  0  0  0
```

d)
```
20 FOR I=1 TO 15
30 A(I)= I
40 PRINT A(I);
run
1 2 3 4 5 6 7 8 9 10
Subscript out of range in 30
ok
```

The system default of 10 is not large enough to handle the entire loop. As a result, the error is displayed.

e)
```
10 DIM A$(15)
20 A$="GOOD AFTERNOON EVERYONE"
30 PRINT A$
run
GOOD AFTERNOON EVERYONE
```

Note

Because the system dynamically allocates string space up to 255 characters, there is no problem initializing a 23-character string constant (line 20) to a character string that has been dimensioned to length 15.

END

END terminates program execution, closes all files, and returns control to the direct mode. An END statement can appear anywhere and often in a program. The END statement is optional at the end of a program.

Syntax

END

Examples

```
10 X=X+1
20 PRINT X
30 IF X>3 THEN END ELSE GOTO 10
run
1
2
3
4
ok
```

When an END statement is placed in the middle of a program, use a CONT statement to resume program execution.

```
50 PRINT "ABC"
60 END
70 PRINT "DEF"
80 END
90 PRINT "GHI"
run
ABC
ok
CONT
```

```
DEF
ok
CONT
GHI
ok
```

ERASE

ERASE eliminates previously dimensioned arrays. After you erase an array, you can redimension it, or the freed array space in memory can be used for other purposes.

Syntax

ERASE array variable [,array variable . . .]

Examples

```
10 DIM A(10)
20 FOR I=1 TO 10
30 PRINT A(I);
40 NEXT I
50 ERASE A
60 DIM A(5)
70 PRINT
80 PRINT
90 FOR I=1 TO 10
100 PRINT A(I);
110 NEXT I
run
0  0  0  0  0  0  0  0  0  0
0  0  0  0  0
Subscript out of range in 90
```

Note

Because A is erased and redimensioned to 5 (lines 50,60), it cannot handle the loop in 80.

ERROR

ERROR defines error codes not intrinsic to N-Basic. The statement can also be used to simulate the occurrence of existing error codes.

Error codes must be an integer greater than 0 and less than 255. N-Basic error codes currently run from 1 to 73. To maintain compatability if more N-Basic error codes are added, use the highest possible value when defining an error code.

If an error statement specifies a code that has not been defined, the message "Unprintable error" is displayed. If the specified code is greater than 255, the message "Illegal function call" is displayed.

Syntax

ERROR code

Example

```
ok
ERROR 14
Out of string space
ok

100 ON ERROR GOTO 130
110 INPUT "WHAT IS YOUR BET";B
120 IF B>5000 THEN ERROR 210
130 IF ERR=210 THEN PRINT "HOUSE LIMIT IS $5000"
140 IF ERL=120 THEN RESUME 110
150 END
run
WHAT IS YOUR BET? 5001
HOUSE LIMIT IS $5000
WHAT IS YOUR BET? 2
ok
```

Note

ERR and ERL are reserved words used in error routines. See Chapter 3.

FIELD

FIELD divides a 256-byte record into data fields for a random disk file. A data field must be identified as a string and cannot exceed 4 characters in length, string identifier ($) included.

Syntax

FIELD file number,field length AS variable, . . .

Example

```
10 OPEN "DATA" AS 1
20 FIELD 1,30 AS NAM$,20 AS ADD$,20 AS
CTY$,5 AS STA$
.
.
.
.
```

Line 20 allocates 30 bytes to the variable NAM$,20 to ADD$,20 to CTY$,5 to STA$.

Note

Variables used in a FIELD statement must not appear to the left of a LET statement or be used in a READ or INPUT statement.

```
WRONG
30 NAM$="GOOF"
```

FILES*

FILES displays the names and lengths of all files residing on a specified disk. Lengths are in units of clusters, normally 8 sectors. If a file is saved by a binary save (system default), it is so designated by a period between the file name and the file type. If saved by an OPEN statement or an ASCII save (SAVE "TEST",A), it is designated by a space in the same position. The default is Drive 1.

Syntax

FILES [drive number]

Example

```
FILES
format.        1          backup.        1
```

```
test1  .asc      1        test2  .        2
demo   .         4        data   dat      1
```

FOR. . .NEXT

The FOR. . .NEXT loop performs a series of instructions a given number of times. FOR opens the loop, increments a variable counter, and sets its lower and upper limits, including increment step (system default 1). NEXT closes the loop and sends control back to its paired FOR statement. This process continues until the upper limit of the loop is reached.

Syntax

FOR variable =x TO y [STEP z]

Examples

a)
```
10 REM SIMPLE
ITERATION
20 FOR I=1 TO 5
30 PRINT I;
40 NEXT I
run
1  2  3  4  5
```

b)
```
10 REM ITERATION
USING STEP
20 FOR I=5 TO 10 STEP 2
30 PRINT I;
40 NEXT I
run
5  7  9
```

c)
```
10 REM ITERATION
USING — STEP
20 FOR I=5 TO 1 STEP
— 1
30 PRINT I;
40 NEXT I
run
5  4  3  2  1
```

d)
```
10 REM NESTED LOOPS
20 FOR I=1 TO 5
30 FOR J=1 TO 5
40 PRINT I+J
50 NEXT J
60 PRINT
70 NEXT I
run
2  3  4  5  6
3  4  5  6  7
4  5  6  7  8
5  6  7  8  9
6  7  8  9  10
```

FORMAT*

FORMAT performs Level 1 formatting on a disk. Disks that are new, formatted on other systems, or improperly formatted must use this command to operate properly. Level 1 formatting prepares a disk for data I/O only. Disk BASIC statements, functions, or commands will not function unless Level 2 formatting is performed. See Chapter 6.

Syntax

FORMAT drive number

Example

```
FORMAT 2
```

Note

Use this statement with extreme caution. Reformatting a disk erases all data previously stored, including the disk operating system.

GET

GET reads a record from a random disk file.

Syntax

GET # file number [,record number]

Examples

```
10 REM PROGRAM TO READ A RECORD
20 REM FROM A RANDOM FILE.
30 OPEN "DATA" AS 1
40 FIELD 1,30 AS NAM$,20 AS ADD$,20 AS CTY$,5 AS
STA$
50 FOR I=1 TO 5
60 INPUT "2 DIGIT RECORD NUMBER";KY%
70 PRINT
80 GET #1,KY%
90 PRINT "NAME" ,NAM$
100 PRINT "ADDRESS", ADD$
```

```
110 PRINT "CITY", CTY$
120 PRINT "STATE", STA$
130 PRINT
140 NEXT I
150 END
run
2 DIGIT RECORD NUMBER ? 4

NAME            CLOWN BOZO THE
ADDRESS         12 BOZO LANE
CITY            BOZO CITY
STATE           MASS

2 DIGIT RECORD NUMBER ? 2

NAME            JUAN DON
ADDRESS         2 LOVERS LANE
CITY            LOVE CITY
STATE           NIRVANA

2 DIGIT RECORD NUMBER ? 1
.
.
.
```

Line 80 demonstrates a GET statement using the option record number. The data was previously stored in file #1 by another program. Line 60 asks for a record number, and that record is read in line 80. If the optional record number is omitted, the program reads the first five records of the file.

GET@

GET@ saves characters or dot graphics within a specified rectangle on the screen to a dimensioned array. You use the coordinates C and R to define the upper-left corner of the rectangle, and c and r to define the lower right corner of the rectangle. Values for C and c range from 0 to the number of characters per line, minus 1. Values for R and r range from 0 to the number of lines per screen, minus 1. All characters within this block will be saved to a previously dimensioned array.

If the G option is specified, both graphics and characters can be saved. When you use this option, the range for C and c is 0 to the number of characters per line *2 minus 1, and the range

for R and r is 0 to the number of characters per line*4 minus 1. The receiving array must be declared as an integer array with a declared subscript greater than the number of characters to be saved.

Syntax

GET@ (C,R) – (c,r), array [,G]

Example

```
10 DIM A% (64) ,B% (100)
20 CONSOLE,,0,1
.
.
.
100 GET@ (10,10) – (17,17) ,A%
200 GET@ (5,5) – (14,14) ,B%
```

GET@ A

GET@ A saves color, dot graphics, and screen attributes within a rectangle specified on the screen to an array. Values of coordinates C and c range from 0 to the number of characters per line minus 1. Values of R and r range from 0 to the number of rows per screen minus 1.

Syntax

GET@ A (C,R) – (c,r), array

Example

```
100 GET@ A (10,10) – (20,20), B%
```

GOSUB/RETURN

GOSUB causes an unconditional break in program execution by transferring control to a designated subroutine. Once program statements of the subroutine are executed, a RETURN statement branches control back to the line immediately following the sending GOSUB statement. There can be more than one RETURN statement referencing the same GOSUB if logic dictates a return from different points of the subroutine. GOSUB statements can be nested. Such nesting is limited only by available memory.

Syntax

GOSUB line number

Example

```
10 PRINT "BEFORE SUBROUTINE J=";J
20 GOSUB 60
30 PRINT
40 PRINT "AFTER SUBROUTINE J= ";J
50 END
60 J=J+5
70 RETURN
run
BEFORE SUBROUTINE J=0

AFTER SUBROUTINE J=5
```

GOTO

GOTO unconditionally branches program execution to a specified line number.

Syntax

GOTO line number

Example

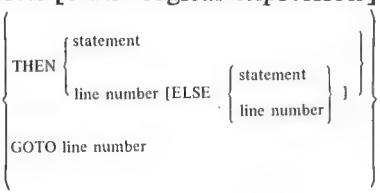
```
10 READ R
20 PRINT "R= ";R;
30 A=3.14*R^2
40 PRINT "AREA= ";A
50 GOTO 10
60 DATA 5,7,12
run
R= 5                AREA= 78.5
R= 7                AREA= 153.86
R= 12               AREA= 452.16
Out of data in 10
```

IF. . .THEN. . .

IF chooses a particular route for program execution based on conditions established in a logical expression.

Syntax

IF logical expression [AND logical expression]



Examples

a) Simple IF Statement

```
10 PRINT TIME$
20 INPUT "TYPE YES OR NO TO CONTINUE";A$
30 IF A$="YES" THEN 10
40 PRINT
50 PRINT "GOODBY"
60 END
run
02:19:16
TYPE YES OR NO TO CONTINUE ? NO

GOODBY
OK
```

The program prints the current time on the system clock repetitively if the test condition in line 30 is met. Program execution continues to the next line (40) if the test condition is not met.

To obtain the same program logic, replace line 30 with the following combinations available to the IF statement.

```
30 IF A$="YES" GOTO 10
30 IF A$="YES" THEN GOTO 10
30 IF A$="NO" THEN 50: ELSE GOTO 10
```

b) AND Option

Use the AND option if more than one condition is required to meet a desired result.

```
10 INPUT "AGE AND INCOME"; AGE,INC
20 IF AGE <65 AND INC>7000 THEN PER=.05:ELSE
PER=.1
30 DIS=300*PER
```

```
40 PRINT
50 PRINT "YOUR DISCOUNT IS "
60 PRINT USING "$##.##";DIS
run
AGE AND INCOME ?64,8000

YOUR DISCOUNT IS
$15.00
ok
```

INPUT

INPUT accepts data from the keyboard during program execution. When encountered, the system prompts you to enter data by displaying a question mark.

Data items must be separated by commas and coincide with the number of variables listed after the INPUT statement. If too few data items are supplied, the system displays two question marks "??" and waits for further input. If too many data items are entered, the message "EXTRA IGNORED" is displayed and execution continues. Data input must correspond to the data types of the variables.

You also have the option of including a prompt string with the INPUT statement to aid in proper data input. When using a prompt string, the system displays the prompt immediately followed by a question mark.

Syntax

INPUT ["prompt string";] variable, variable . . .

Examples

a)
```
10 INPUT "INPUT A NUMBER";X
20 PRINT X " SQUARED IS"; X^2
run
INPUT A NUMBER? 12.45
12.45 SQUARED IS 155.003
ok
```

b)
```
10 INPUT "INPUT BASE AND HEIGHT ";B,H
20 AREA= (B/2) *H
```

```
30 PRINT "THE AREA OF THE RIGHT TRIANGLE
IS ";AREA
run
INPUT BASE AND HEIGHT ? 5,5
THE AREA OF THE RIGHT TRIANGLE IS 12.5
ok
```

INPUT

INPUT# reads a record from a previously created sequential file and initializes its variable list to the appropriate record items.

Variables in an INPUT# statement must be identical in data type to the stored record items. For example, the variable X$ cannot be initialized to a record item stored as an integer.

INPUT# reads an image of data on disk as it was saved by the PRINT# statement. Improper results may be caused by improper storage.

Syntax

INPUT# file number, variable, variable, . . .

Example

```
10 OPEN "list.dat" FOR INPUT AS #1
20 IF EOF (1) THEN END
30 INPUT #1,N$,S$,D$
40 PRINT N$
50 PRINT S$
60 PRINT D$
70 PRINT
80 GOTO 20
run
JIM BEAM
PUBLIC RELATIONS
10/10/80

HULK
DEMOLITION
12/21/80
```

POPEYE
RECREATION
9/5/80

BO DEREK
SALES
10/31/80

Note

The preceding data was previously stored using the PRINT# statement.

A file number specified as −1 (INPUT FILE #−1. . .) is read from cassette.

KEY

KEY initializes the programmable function keys displayed at the bottom of the screen. Of the ten keys available, five are displayed in lowercase mode (default) and five are displayed in uppercase mode. Key numbers 1 through 5 are reserved for lowercase, 6 through 10 are reserved for uppercase. The maximum length of a string is 15 characters, although only 12 characters are displayed in 80-character mode and 6 in 40-character mode.

Function keys are programmable in both standard and control characters. Control characters that cannot be entered from the keyboard can be specified using the function CHR$ (n) and appended to the string using a plus (+) sign. CHR$ (n) returns the ASCII character whose code is decimal n. See Appendix D.

Syntax

KEY key number,"string"

Examples

KEY 1,"load"	The load command appears in the first function key display box at the bottom of the screen.

KEY 10,"list" +CHR$ (13)	Function KEY 10 (uppercase) is programmed to list (list carriage return).

KEY LIST

KEY LIST displays a complete list of strings assigned to the function keys.

Syntax

KEY LIST

Examples

```
KEY LIST
H┬                time$
auto              key
go to             print
list              list
run               cont
ok
```

KILL

KILL deletes a specified file from any mounted disk.

Syntax

KILL "[drive:] filename"

Examples

KILL "DATA.1"	The file "DATA.1" is deleted from the disk mounted on Drive 1 (default).
KILL "2:DATA.1"	"DATA.1" is deleted from the disk mounted on Drive 2.

LET

LET assigns a value or the value of an expression to a variable. Its use is optional.

Syntax

[LET] variable=value

Example

The following examples are equivalent.

```
10 LET D$="HELLO"          10 D$="HELLO"
20 LET A=100               20 A=100
30 LET B=8^2               30 B=8^2
40 LET TOTAL=A+B           40 TOTAL=A+B
```

LFILES*

LFILES lists all files of a disk mounted on a specified drive to a printer. Default is Drive 1.

Syntax

LFILES [drive number]

Example

LFILES 2	All files on Drive 2 are listed to the printer.

LINE

In its simplest form, LINE draws a line, displayed on the screen, between two user-defined points. Three syntaxes are available.

Syntax 1

LINE screen line,function code

You use this syntax to specify various attributes to a line displayed on the screen. The function codes that assign these attributes are listed in Table 2-3.

Table 2-3 Line Function Codes for Syntax 1

CODE	FUNCTION
0	Normal
1	Blinking

2	Reverse field
3	Reverse field with blinking

Specified line numbers must range between 0 and the number of lines per screen minus 1. Because a LINE statement using Syntax 1 will not function in black and white mode, first execute a CONSOLE statement to put the monitor in color mode.

Example 1

```
.
.
.
50 CONSOLE 0,19,0,1
60 PRINT CHR$ (12)
70 LINE 3,2
80 LOCATE 0,3: INPUT "WHAT IS YOUR BET ";B
.
.
.
```

Note

Because of line 70, all characters that appear in line 3 are displayed in reverse field.

Syntax 2

LINE (C,R)–(C,R),"string" [,function code] [,B[F]]

You use this syntax to draw lines between specified points. Column must range between 0 and the number of characters per line minus 1. Row must range between 0 and the number of lines per screen minus 1. For example, using the default screen width of WIDTH 40,20, the range of C is 0 to 39 and the range of R is 0 to 19.

You use the string option to specify the character used in drawing a line. A string option specified as "◆" draws a line of diamonds.

The function code option assigns various visual attributes to the screen. Function codes and attributes are listed in Table 2-4.

Table 2-4 LINE Function Codes for Syntax 2

CODE	COLOR MODE	BLACK AND WHITE MODE
0	Black	Normal
1	Blue	Secret
2	Red	Blink
3	Magenta	Secret
4	Green	Reverse Field
5	Cyan	Reverse Field-Secret
6	Yellow	Reverse Field-Blink
7	White	Reverse Field-Secret

The B option draws the border of a rectangle using the coordinates of C and R as the upper left and lower right corners of the rectangle. Specify F to fill the rectangle.

Example 2

LINE (15,4)–(24,14),"◆",B

See Figure 2-1A.

Syntax 3

LINE (C,R)–(C,R), point set [,function code] [,B[F]]

This syntax uses dot graphics to draw a line between specified points. The range of C is 0 to the number of characters per line *2 minus 1. The range of R is 0 to the number of lines per screen*4 minus 1. For example, WIDTH 80,25 gives C a range of 0 to 159 and R a range of 0 to 99.

There are two possible values for the point set option, PSET and PRESET. To draw a line, use PSET; to erase the line use PRESET. Successively drawing and erasing a particular line at various intervals across the screen simulates movement.

See Syntax 2 for an explanation of the remaining options ([function code] [,B[F]]).

Example 3

```
10' PROGRAM TO ROTATE A LINE
20' 360 DEGREES 10 SUCCESSIVE TIMES
30 FOR J=1 TO 10
40 PRINT CHR$ (12)
50 FOR I=1 TO 20
60 LINE (19+I,41-I)-(61-I,39+I),PSET
70 LINE (19+I,41-I)-(61-I,39+1),PRESET
80 NEXT I
90 FOR I=1 TO 19
100 LINE (40+I,20+I)-(40-I,60-I),PSET
110 LINE (40+I,20+I)-(40-I,60-I),PRESET
120 NEXT I
130 LINE (60,40)-(20,40),PSET
140 NEXT J
```

See Figure 2-1B.

```
LINE (139,80)-(150,87),PSET,2,B
```

See Figure 2-1C.

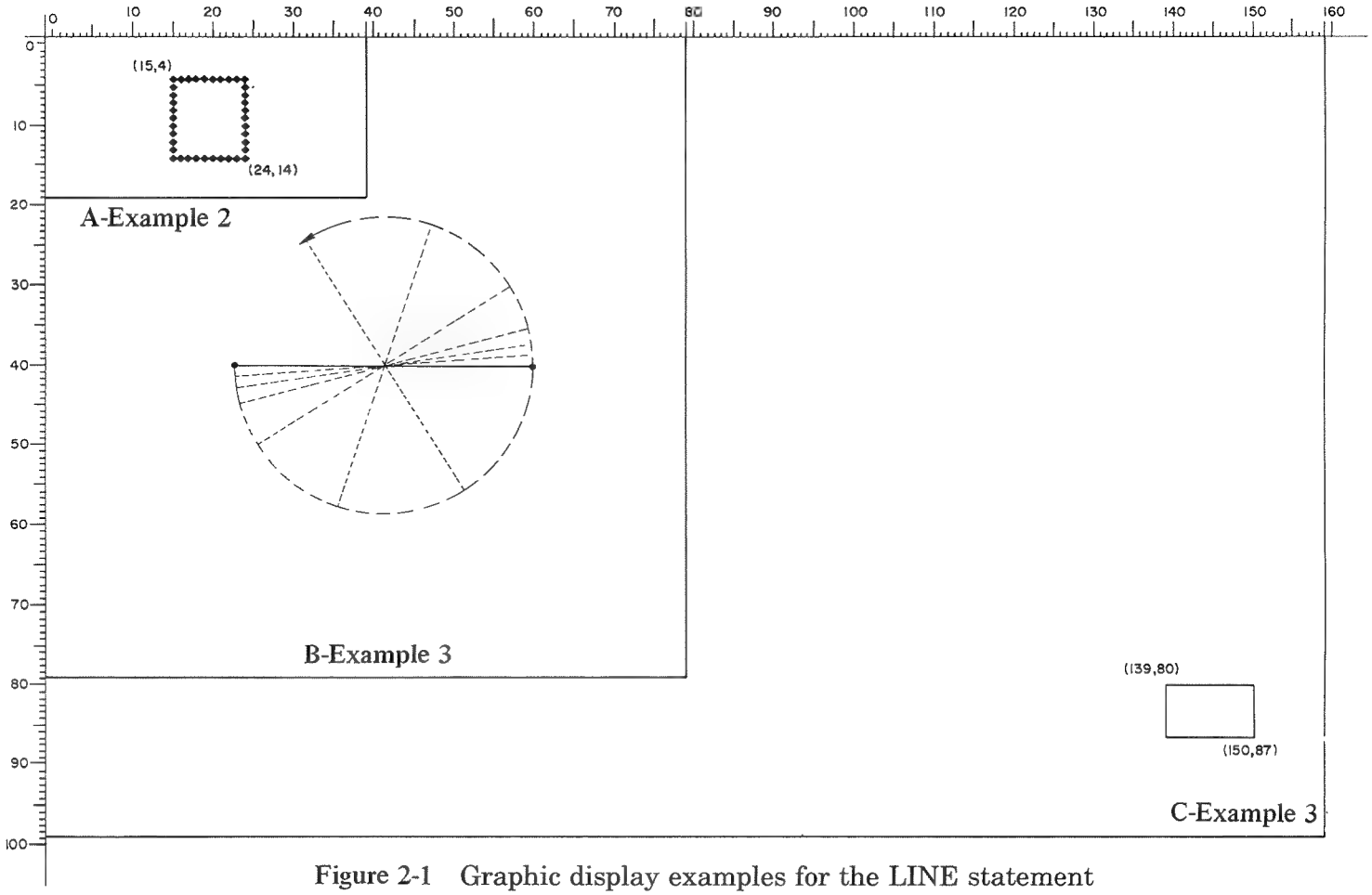


Figure 2-1 Graphic display examples for the LINE statement

LINE INPUT

LINE INPUT initializes a string variable to an entire line of up to 255 characters, including delimiters. LINE INPUT does not prompt a question mark (?) as the INPUT statement unless it is included in the prompt string.

Syntax

LINE INPUT ["prompt string";] string variable

Example

```
10 OPEN "OUT" FOR OUTPUT AS 1
20 LINE INPUT "CUSTOMER INFORMATION ?";C
$
30 PRINT #1,C$
.
.
.
```

Note

The LINE INPUT statement in line 20 prompts CUSTOMER INFORMATION? and waits for input. Use a Control-C to escape the LINE INPUT statement and return the system to direct mode. Conversely, enter CONT in direct mode to resume program execution at the LINE INPUT statement.

LINE INPUT#

LINE INPUT# reads a line from disk as it was saved by the LINE INPUT statement. LINE INPUT# reads all characters in a sequential file up to a carriage return, ignoring all previous delimiters.

Syntax

LINE INPUT# file number, variable

Examples

```
10 PRINT CHR$ (12)
20 OPEN "OUT" FOR OUTPUT AS 1
30 FOR I=1 TO 3
40 LINE INPUT "Customer information ?";C$
50 PRINT #1,C$
60 NEXT I
70 CLOSE 1
75 PRINT
80 OPEN "OUT" FOR INPUT AS 1
90 FOR I=1 TO 3
100 LINE INPUT#1,C$
110 PRINT C$
120 NEXT I
130 CLOSE 1
run
Customer information ? LINDA JONES 12.2,
3 OHIO
Customer information ? TOM JONES 12.3,4
FLORIDA
Customer information ? SAM JONES 12.4,5
MAINE

LINDA JONES 12.2,3 OHIO
TOM JONES 12.3,4 FLORIDA
SAM JONES 12.4,5 MAINE
```

LIST*

LIST lists all or part of a program currently in memory. Use the ESC or STOP keys to halt a listing at any point of the program.

The ESC key halts a listing, which remains in indirect mode. The listing can be restarted simply by pressing any character key, including a FUNCTION key, RETURN key, or the space bar.

The STOP key halts a listing and returns system control to direct mode. For a further listing, you must reenter one of the various list options.

Syntax

LIST [line number] [{-,} line number]

Examples

LIST	Lists the entire program currently in memory.
LIST 500	Lists line 500.
LIST 150-	Lists all lines from line 150 to the end of the program.
LIST -150	Lists all lines from the beginning of the program to line 150.
LIST 50-100	Lists all lines from 50 to 100 inclusive.
LIST.	Lists a line that caused an error and halted program execution.

Note

A comma (,) can be substituted for a hyphen (-).

LLIST*

LLIST lists a complete or partial listing of the program currently in memory to the printer. All combinations available to the LIST statement are available to LLIST, with the exception of the period option (LIST.).

Use the STOP key to stop printing a listing. You cannot use the ESC key with LLIST.

Syntax

LLIST [line number] [{-,} line number]

Examples

LLIST 50-100	Lists lines 50 to 100 inclusive to the line printer.

LOAD*

LOAD loads a file on disk into memory, closes all open files, and deletes all program lines and variables currently residing in memory.

LOAD used with the R option (load and run) deletes program lines and variables but does not close opened data files. Use this option to chain programs too large to fit into available memory.

You use the option drive number to specify a drive other than Drive 1 (default).

Syntax

LOAD "[drive number:] filename" [,R]

Examples

LOAD "GAME"	Loads the file GAME into memory from Drive 1.
LOAD "GAME" ,R	Loads and runs the file GAME.
LOAD "2:GAME"	Loads the file GAME from Drive 2.

LOCATE

LOCATE positions the cursor at a specified location on the screen. The coordinates of Column and Row must be within a range of 0 to 254. If a specified coordinate is greater than the current screen size, the system defaults to the last valid location. For example, LOCATE 250,10 in 40-character mode defaults to LOCATE 39,10.

Specify a 0 for the option cursor switch to turn off the cursor.

Syntax

LOCATE column,row [,cursor switch]

Example

```
30 LOCATE 10,10
40 PRINT "Hello"
```

LSET, RSET

Before data of a random file can be stored on disk, move it into the defined fields of a random file buffer. Use LSET and RSET for this movement.

LSET left justifies data placed in a field; RSET right justifies it. The system pads a string that is too short to fit in a particular field with spaces and truncates a string that is too long.

You can only move strings into a random file buffer. Convert numeric data to a string before using LSET and RSET. The functions that perform these conversions are MKI$, MKS$, and MKD$. For more information concerning these functions, see Chapter 3.

Syntax

LSET field name=string variable
RSET field name=string variable

Example

```
10 OPEN "bozo" AS #1
20 FIELD #1,20 AS NAM$,10 AS INC$
30 INPUT "How many entries ";j
40 FOR I=1 TO J
50 INPUT "Name ";N$
60 INPUT "Salary "; INC
70 INPUT "2 digit key ";KY%
80 LSET NAM$=N$
90 RSET INC$=MKS$ (INC)
100 PUT #1,KY%
110 NEXT I
120 CLOSE 1
130 END
run
How many entries ? 1
Name ? Joe Shmoo
Salary ? 18000
```

Line 20 defines the random file buffer as:

Lines 80 and 90 move the entered data into the random file buffer.

The data is now ready to be stored on disk using the PUT statement in line 100.

Note

N-Basic performs the string to numeric conversion in line 90 (MKS$ (INC)) before the data is right justified (RSET) to the buffer.

MERGE*

MERGE merges a program on disk with a program currently in memory. Use ASCII format to save the program on disk. Statements with identical line numbers are overwritten by the program called from disk.

The drive option specifies which disk the program is stored on. Default is Drive 1.

Syntax

MERGE "[drive:] filename"

Example

The following lines have been stored on Disk 2 using ASCII format under the name "DEMO".

```
5 PRINT "  MERGE DEMO
30 PRINT "And now it is
```

```
40 PRINT "Time to make
50 PRINT "Payment
```

The next series of lines are entered at the keyboard.

```
10 PRINT "We have gathered
20 PRINT "And we have spent
30 PRINT "And we are broke

MERGE "2:DEMO"
ok
run

MERGE DEMO
We have gathered
And we have spent
And now it is
Time to make
Payment
```

MOTOR

MOTOR controls the motor of a cassette tape recorder, which can be attached to the PC-8000. For MOTOR to work properly, hook up the wires of the cassette correctly and set its play button to ON. Connect the three color coded wires of the cassette attachment cable as follows.

COLOR	POSITION
Black	Remote Jack
White	Earphone Jack
Red	Microphone Jack

Specify a 0 for the switch option to turn the motor off. Any value greater than 0 turns the cassette motor on.

MOTOR used without the switch option turns the cassette motor on or off depending on its last state.

Syntax

```
MOTOR [switch]
```

Examples

MOTOR 1	Turns the cassette motor on.
MOTOR 0	Turns the cassette motor off.

MOUNT*

Execute MOUNT before using a disk. When MOUNT is executed, N-Basic reads the disk File Allocation Table (FAT) into memory and checks for errors. If the FAT is free from errors, the disk is mounted. If errors are detected, the system reads one or both of the back-up tables. The message "x copies of allocation bad on drive y" is displayed, indicating a new copy of the disk should be made. If all FATs are bad, the message "bad allocation tables" is displayed. This error is unrecoverable and the disk cannot be mounted.

Syntax

MOUNT drive number,drive number . . .

Example

MOUNT 1,2	The disks on Drives 1 and 2 are mounted.

NAME*

NAME renames a file already stored on disk.

Syntax

NAME "old filename" AS "new filename"

Example

```
NAME "TEST" AS  "TEST2"
The old file name TEST is changed
to TEST2.
```

NEW*

NEW deletes the program currently in memory and clears all variables. Use NEW to clear memory before entering a new program.

Syntax

NEW

Example

```
NEW
ok
```

ON ERROR GOTO

ON ERROR GOTO sets up an error trap for any errors that may occur during program execution. When an error is encountered, program control passes to the error routine specified by the GOTO statement.

ON ERROR GOTO 0 disables the ERROR trap routine. Subsequent errors are displayed in their normal fashion. If an ON ERROR GOTO 0 statement is executed in an error trap routine, the appropriate system error message is displayed and control returns to direct mode.

Syntax

ON ERROR GOTO line number

Example

```
10 ON ERROR GOTO 100
20 FOO I=1 TO 3
30 PRINT I
40 NEXT I
45 STOP
100 PRINT "YOU MADE A MISTAKE"
110 END
run
YOU MADE A MISTAKE
ok
```

Note

The incorrect line would have normally displayed the message "Syntax error in 20".

ON GOSUB/ON GOTO

ON GOSUB/ON GOTO branches program flow to one of several specified line numbers. The branch is contingent on the value of the variable expression. For example, if the value of the expression is 3, control branches to the line number specified by the third value in the line list.

Syntax

ON expression { GOSUB / GOTO } line,line,line . . .

Example

```
10 PRINT "ENTER A 1-IF YOU WISH TO WITHD
RAW 2-TO DEPOSIT 3-TO WITHDRAW FROM C
HECKING ACC. 4-TO DEPOSIT CHECKING ACC.
":INPUT A
20 ON A GOTO 50,70,90,100
.
.
.
```

Note

A 2 entered for the variable A branches control to line 70. A 3 branches control to line 90. A negative number for the variable A causes the error message "Illegal function call in line 20". A value greater than the amount of line numbers in the list, in this example a value greater than 4, branches control to the next logical line.

OPEN

OPEN opens a random or sequential data file on disk for input or output. You must specify a file name and a file number. The file name is the name the file will be stored under. It must not exceed 6 characters in length excluding file type identifier.*

*See SAVE for an explanation on file type identifiers.

The file number serves as a temporary abbreviated name usually used in place of a file name. In this manner, a file opened as (OPEN "data. dat" AS#1) can be referenced as PUT#1 or GET#1. The file number is valid until the file is closed. If reopened, it is not necessary to use the same file number.

The choice of a file number is limited by the value that is entered for the prompt "How many files" when booting up disk BASIC. A 3 entered for this prompt limits the values of file number to 1, 2, and 3.

The syntax for OPEN is slightly different for random and sequential files. The three modes that can be used for sequential files are INPUT, OUTPUT, and APPEND (see Syntax 1).

- INPUT — positions the file pointer at the beginning of a file and is used primarily for sending data to memory from disk.
- OUTPUT — positions the pointer at the beginning of a file and is used primarily for writing data to a disk.
- APPEND — positions the pointer at the end of the file and is used for adding additional data to a file.

The OPEN statement for random files is less complicated. One mode is used for reading, writing, and appending data to a disk (see Syntax 2).

Syntax 1 (Sequential Files)

OPEN "filename" [FOR mode] AS [#] filenumber

Example 1

```
10 OPEN "data.dat" FOR OUTPUT AS #1
```

Line 10 opens the file "data.dat" for data output to the disk. The file is assigned file number 1.

Note

If you omit the option [FOR mode], the system defaults to output mode.

Syntax 2 (Random Files)

OPEN "filename" AS [#] filenumber

Example 2

```
10 OPEN "data.dat" AS 2
```

Opens the data file "data.dat" under the temporary file number 2.

OUT

OUT sends a byte of information to a machine port. The integer "I" is the port number and J is the data to be transmitted.

Syntax

OUT I,J

Example

```
100 OUT 32,100
```

PEEK AND POKE

You use PEEK to read a particular byte of information in memory. The range of the integers for PEEK is the same as that for POKE.

POKE writes a byte of information to memory. The integer "I" is the memory address that must be in the range of 0 to 65535. A specified J is the data to be written to memory that must be in the range of 0 to 255. I and J can be hex, octal or decimal values.

Syntax

POKE I,J
PEEK (I)

Example

```
10 POKE &H5A00,&HFF
20 PRINT PEEK (&H5A00)
```

POINT

POINT checks whether a dot is set at a specified location. POINT returns value of minus 1 if a dot is set at the specified location, or 0 if it is not set.

Syntax

POINT (column, row)

Example

```
200 IF POINT (50,65)= 0 THEN RETURN
300 PRESET (50,65)
```

PRINT

PRINT displays the values of numeric and string variables, expressions, and user-defined prompts. The position these items are displayed in is determined by the punctuation used to separate the list of items to be printed.

The screen width is divided into 14 space zones. A comma used to separate items places the data every 14 spaces. A semicolon places data every other space.

A PRINT statement that is terminated by a comma or a semicolon causes the next PRINT statement to begin printing on the same line, spaced accordingly. A PRINT statement not terminated by a comma or a semicolon is followed by a carriage return.

A question mark can be used as an abbreviated form of PRINT.

Syntax

PRINT ["prompt" [;] item,item,item. . .]

Examples

In the following examples, line 20 is altered to illustrate various punctuation combinations. Displays are shown in the default screen width of 40 characters per line, 20 lines per screen.

a)
```
10 FOR I=1 TO 6
20 PRINT I
30 NEXT I
run
1
2
3
4
5
6
```

b)
```
20 PRINT I;I+1;
run
1 2 2 3 3 4 4 5 5 6 6 7
```

c)
```
20 PRINT I;I+1,
run
1  2        2  3
3  4        4  5
5  6        6  7
```

d)
```
20 PRINT I,I+1;
run
1           2  2
3  3        4  4
5  5        6  6
7
```

e)
```
20 PRINT I,I+1,
run
1           2
2           3
3           4
4           5
5           6
6           7
```

```
10 INPUT X
20 PRINT X "SQUARED IS "X^2" AND";
30 PRINT X "CUBED IS "X^3
40 PRINT
50 GOTO 10
run
? 8
8 SQUARED IS 64 AND 8 CUBED IS 512

? 4
4 SQUARED IS 16 AND 4 CUBED IS 64

?
```

Note

A PRINT statement used alone skips a line.

PRINT#

PRINT# writes an image of data as it is displayed on the screen to a disk.

Syntax

PRINT# filenumber, [USING "format string";] variable, variable, . . .

Examples

For this example, a program writes data to a file on disk.

```
10 REM PROGRAM -WRITE
20 OPEN "test" FOR OUTPUT AS #1
30 INPUT "NAME ";N$
40 INPUT "CLAIM NO.";C$
50 PRINT# 1,N$;C$
60 CLOSE #1
run
NAME ?SAM SPADE
CLAIM NO. ?123-321
ok
```

The program "READ" reads the newly created file.

```
10 REM PROGRAM-READ
20 OPEN "test" FOR INPUT AS #1
30 INPUT #1,N$,C$
40 PRINT N$,C$
50 CLOSE #1
run
Input past end in 20
```

The error message is caused by the program "WRITE". The PRINT# statement in line 40 of "WRITE" stores N$ and C$ as one. Hence, instead of SAM SPADE 123-321 being stored as two separate strings, it is actually stored as one SAM SPADE123-321. The INPUT# statement of "READ", trying to input two strings and finding only one, prompts the error message.

To correct the problem, insert a delimiter between the input strings of the "WRITE" program.

```
40 PRINT#1 N$;",";C$
```

The program READ now yields the proper results.

```
run "READ"
SAM SPADE 123-321
ok
```

If a string to be written to disk contains delimiters of its own (commas, semicolons, significant leading blanks, carriage returns or line feeds), the string must be surrounded by explicit quotation marks. Use the function CHR$(n), which returns the character for the ASCII code (n). The ASCII code for quotation marks (") is 34.

To store ("SAM SPADE, DETECTIVE" 123-321), make the following changes to the program "WRITE".

```
35 DEL$=CHR$ (34)
40 PRINT#1=DEL$;N$;DEL$;C$
```

```
list

10 OPEN "test" FOR OUTPUT AS #1
20 INPUT "NAME ";N$
30 INPUT "CLAIM NO. ";C$
35 DEL$=CHR$ (34)
40 PRINT#1, DEL$;N$;DEL$;C$
50 CLOSE #1

run
NAME ? "SAM SPADE, DETECTIVE"
CLAIM NO. ? 123-321
ok

run "READ"
SAM SPADE, DETECTIVE
123-321
ok

10 OPEN "test" FOR OUTPUT AS #1
20 A=9.5:B=8.32:C=7:D=6.0
```

```
30 PRINT#1,A;B;C;D
40 CLOSE #1
run
ok

10 OPEN "test" FOR INPUT AS #1
20 INPUT #1,A,B,C,D
30 PRINT A;B;C;D
40 CLOSE #1
run
9.5  8.32  7  6
```

The syntax (PRINT# filenumber, USING "format string") controls the format of data to be placed on the disk. See PRINT USING for further details.

Example

```
PRINT #1, USING "##.## ";A,B.C
```

Note

A file number specified as a "−1" writes data to a cassette tape.

PRINT USING

PRINT USING prints data to a formatted field. To define the fields, use the various reserve symbols for string and numeric data shown below.

String Data

There are two formatting characters that can be used to print string data to a field:

!	specifies that only the first character of a given string is to be printed.
"&+spaces&"	displays a specified number of characters in a given string, determined by two plus the number of spaces between the delimiter "&".

Syntax 1

PRINT USING "format symbol"; string variable;string variable; . . .

Example 1

```
10 A$="good"
20 B$="karma"
30 PRINT USING "!";A$;B$
40 PRINT USING "!&    &";A$;B$
50 PRINT USING "&       &";A$;B$;"!!"
run
gk
gkarma
good karma !!
ok
```

Note

The PRINT USING statement left justifies and pads with spaces data too short to fit a specified field and truncates a field too long.

Numeric Data

There are several formatting characters that can be used to print numeric data in a field.

Syntax 2

PRINT USING "format string";variables,variables . . .

Example 2

#	Represents each digit of a field. An implied decimal point can be used at any position of the field. Zeros are placed in all unused positions right of the decimal point and unused positions left of the decimal are filled with blanks. Fractions are displayed with at least one zero preceded by blanks left of the decimal. Data to be

placed in a field must contain its own decimal point for proper results. Numbers are rounded when necessary.

```
PRINT USING "###.##";1.,.2,1.2
1.00 0.20 1.20
```

+ When at the beginning of a format string, prints the sign, plus or minus, of the value.

– When at the end of a format string, displays negative values with a trailing minus sign.

```
PRINT USING "+###.##";111.22,222.33,
-444.44
+111.22 + 222.33 - 444.44

PRINT USING "###.##- ";111.22,222.33,
-444.44
111.22 222.33 444.44-
```

** When at the beginning of a format field, fills all leading spaces with asterisks. They are also considered two digit positions.

```
PRINT USING "**##.## ";12.39,
-0.9,765.1,1.0
**12.40 **-0.90 *765.10 ***1.0
```

, When used immediately left of a decimal point, places a comma every third digit left of the decimal. When placed at the end of a format string, the comma is displayed as part of the defined field.

```
PRINT USING "#########,
.##  ";6000.00,1170000.20
6,000.00 1,170,000.20

PRINT USING "#####.##, ";6000.00,
5000.125
6000.00, 5000.13
```

^^^^ When placed at the end of a format string specifies exponential notation. The carats assign space for E+nn to be displayed.

```
PRINT USING "###.##^^^^ ";123.56
12.36E+01
```

% is displayed by the system to warn that a value is greater than a specified field.

```
PRINT USING "##.##;123.22
%123.22
```

PSET, PRESET

PSET and PRESET set or clear a dot at a specified location on the screen. The dot is set by the coordinates column and row. Column can be a value from 0 to the (number of characters per line * 2) minus 1, and the value of row can be 0 to the (number of lines per screen * 4) minus 1.

For example, WIDTH 80,20 limits the range of column from 0 to 159 and row from 0 to 79.

Use the option function code to specify a color in color mode or a screen attribute in black and white mode. Codes, colors, and options are listed in Table 2-5.

Table 2-5 PSET, PRESET Option Codes

CODE	COLOR (COLOR MODE)	ATTRIBUTES (BLACK AND WHITE MODE)
0	Black	Normal
1	Blue	Secret
2	Red	Blink
3	Magenta	Secret
4	Green	Reverse Field
5	Cyan	Reverse Field-Secret
6	Yellow	Reverse Field-Blink
7	White	Reverse Field-Secret

Syntax

{ PSET / PRESET } (column, row ,function code)

Example

```
40 PSET (20,20,2)
```

Note

PSET and PRESET will not always function properly out of the graphic mode.

PUT

PUT writes the contents of a random buffer to a random disk file. The file number is the number that the file was opened under. Record number indicates the record that the contents of the buffer are to be written to. If you omit record number, the contents of the random buffer are written to the next available record of the random file. See section on Random Files.

Syntax

```
PUT #file number [,record number]
```

Example

```
10 OPEN "data.dat" AS #1
20 FIELD #1,30 AS NAM$,20 AS ADD$,20 AS CTY$,
5 AS STA$
.
.
.
130 PUT #1,KY%
.
.
.
```

PUT@

PUT@ displays characters and dot graphics previously saved by a GET@ statement in a specified area of the screen. You define the rectangular display area by specifying the column and row locations of the upper-left and lower-right corners. Optionally, you can specify screen color or attributes for the display area.

The C, R specifies the upper-left corner, and the c, r specifies the lower-right corner.

With character display, values of C and c range from 0 to the number of characters per line minus 1. Values of R and r range from 0 to the number lines per screen minus 1. You can set attributes in the same way as the attributes in the COLOR statement (see "COLOR") with character display.

With dot graphic display, values of C and c range from 0 to the number of characters per line *2, minus 1. Values of R and r range from 0 to the number of lines per screen *4 minus 1. You can set a condition of PSET, PRESET, OR, AND, NOT, or XOR only.

Use the PUT@ A statement (not PUT@) to display arrays saved using a GET@ A statement.

Syntax

PUT@ (C,R) – (c,r), array, { attribute / condition }

Example

```
100 DIM A% (256), B% (100)
400 GET@ (0,0) – (15,15), A%, G
700 GET@ A (10,10) – (19,19), B%
800 PUT@ (35,40) – (50,55), A%, XOR
810 PUT@ A (30,13) – (39,21), B%
```

PUT@ A

PUT@ A displays characters and dot graphics previously saved by a GET@ A statement in a rectangular area of the screen. You define the rectangular display area by specifying the column and row locations of the upper-left and lower-right corners.

For character or dot graphic display, values of C and c range from 0 to the number of characters per line, minus 1. Values of R and r range from 0 to the number of lines per screen, minus 1.

Unlike the PUT@ statement, no attribute or condition can be displayed. Use the PUT@ statement (not PUT@ A) to display arrays saved by a GET@ statement.

Syntax

PUT@ A (C,R) – (c,r), array

READ

READ initializes variables to the data items of a DATA statement on a one-to-one basis. The READ statement variables can be of any data type, but must agree with the data type of the items in the DATA statement.

A single READ statement can access several DATA statements; several READ statements can access one DATA statement.

If the number of variables in a READ statement exceeds the number of data items, the message "Out of DATA in nn" is displayed. If the number of DATA items exceeds the variables of a READ statement, they are ignored.

Syntax

READ variable, variable, . . .

Examples

a)
```
10 FOR I=1 TO 5
20 READ A
30 PRINT A;
40 NEXT I
50 DATA 1,2,3,4,5,7
run
1  2  3  4  5
```

b)
```
10 FOR I= 1 TO 5
20 READ A,B
30 PRINT A;B
40 NEXT I
50 DATA 1,2,3,4,5,
6,7,8,9,10
run
1  2
3  4
5  6
7  8
9  10
```

c)
```
5 WIDTH 80
10 PRINT "CITY","STATE"," ZIP"
20 READ C$,STA$,Z
30 DATA ELKGROVE, ILL, 60007
40 PRINT C$,STA$,Z
run
CITY          STATE   ZIP
ELKGROVE      ILL     60007
```

Items in a DATA statement are read only once from start to finish. To reread the DATA from the beginning, execute a RESTORE command before the next READ statement.

```
5 WIDTH 80
10 PRINT "CITY","STATE"," ZIP"
20 READ C$,STA$,Z
30 DATA ELKGROVE, ILL, 60007
40 PRINT C$,STA$,Z
50 RESTORE
60 PRINT
70 READ C$
80 PRINT "THE CITY OF";C$;"GREETS YOU
run
CITY          STATE   ZIP
ELKGROVE      ILL     60007

THE CITY OF ELKGROVE GREETS YOU.
```

REM or '

REM is a nonexecutable statement used for explanatory remarks concerning a program. Remarks can follow a program line if you precede the remark with an apostrophe. You can also use the apostrophe as an abbreviated form of REM.

Syntax

REM [remark]

Example

```
4 REM PROGRAM NAME "RANLIS"
5 ' PROGRAM TO CREATE A RANDOM FILE
10 OPEN "data" AS #1
20 FIELD #1,30 AS NAM$,20 AS ADD$,20 AS CTY$,5 AS
STA$
30 INPUT "HOW MANY ENTRIES ";J
40 FOR I=1 TO J
50 INPUT "NAME ";N$
60 INPUT "ADDRESS ";A$
70 INPUT "CITY ";C$
80 INPUT "STATE ";S$
90 INPUT "2 DIGIT  KEY ";KY%   ' KY% IS THE
100 REM                        ' RECORD NUMBER
110 LSET NAM$=N$
                               ' LINES 110 THROUGH 140
120 LSET ADD$=A$               ' LEFT JUSTIFY DATA
                               ' IN THE RANDOM
130 LSET CTY$=C$               ' FILE BUFFERS
140 LSET STA$=S$
150 PUT #1,KY%
160 NEXT I
170 CLOSE 1
180 END
```

REMOVE

REMOVE updates the File Allocation Table (FAT) on a disk before it is physically removed from a drive. When executed, three copies of the FAT currently residing in memory are made to the FAT area of the disk.

Always execute the REMOVE command before taking a disk out of a drive. Failure to do so causes two problems.

a. The FAT on the disk is not updated or checked for errors if file allocation is changed.

b. The FAT of the previous disk is copied to the FAT area of the current disk, making file accessing impossible. Hence, all files on the new disk are effectively destroyed,

because the old FAT does not reflect the actual file locations on the new disk.

Syntax

REMOVE [drive, drive, . . .]

Examples

REMOVE 1,2	The disks in Drives 1 and 2 are removed.

Note

Using REMOVE without the drive option removes the disks on all drives.

RENUM

RENUM renumbers program lines and changes all line number references following GOTO, GOSUB, THEN, ON. . . GOTO, ON. . .GOSUB and ERL.

Syntax

RENUM [new number] [,old number] [,increment]

Examples

a) RENUM — Renumber the entire program starting at line 10 at increments of 10.

BEFORE	AFTER
1 PRINT 1	10 PRINT 1
2 PRINT 2	20 PRINT 2
3 PRINT 3	30 PRINT 3
4 PRINT 4	40 PRINT 4

b) RENUM 2 — Renumber the entire program beginning at line 2 at 10-line increments.

BEFORE	AFTER
10 PRINT 1	2 PRINT 1
20 PRINT 2	12 PRINT 2

```
30 PRINT 3                    22 PRINT 3
40 PRINT 4                    32 PRINT 4
```

c) RENUM 100,,50 — Renumber the entire program beginning at line 100 at increments of 50.

```
   BEFORE                     AFTER
2 PRINT 1                     100 PRINT 1
12 PRINT 2                    150 PRINT 2
22 PRINT 3                    200 PRINT 3
32 PRINT 4                    250 PRINT 4
```

d) RENUM 300,150,50 — Change line 150 to 300 and increment all following lines by 50.

```
   BEFORE                     AFTER
100 PRINT 1                   100 PRINT 1
150 PRINT 2                   300 PRINT 2
200 PRINT 3                   350 PRINT 3
250 PRINT 4                   400 PRINT 4
```

RESTORE

RESTORE resets the items of a DATA statement so they can be reread by a READ statement. It also provides the option of choosing the particular DATA statement to be read.

Syntax

RESTORE [line number]

Example

```
10 READ A,B,C
20 PRINT A;B;C
30 READ A,B,C,D,E,F
40 RESTORE 80
50 READ G,H,I
60 PRINT A;B;C;D;E;F;G;H;I
70 DATA 1,2,3,4,5,6,7,8
80 DATA 9,10,11,12
```

```
run
1  2  3
4  5  6  7  8  9  9  10  11
```

RESUME

Use RESUME to continue program execution after performing an error recovery procedure. Its options are as follows.

- RESUME [0] — resumes execution at the statement that caused the error.
- RESUME NEXT — resumes program execution at the line immediately following the faulty line.
- RESUME line number — resumes program execution at the specified line number.

Syntax

RESUME [Ø | NEXT | line number]

Example

```
10 ON ERROR GOTO  80
20 A = 42
30 PRIN A
40 B=A*2+3
50 PRINT
60 PRINT B
70 END
80 PRINT "ERROR ERROR "
90 RESUME NEXT
100 PRINT "CORRECT PROGRAM BEFORE
CONTINUING."
run
ERROR ERROR

 87
```

Because the NEXT option is used in line 90, program execution resumes at line 40. If RESUME 0 were used, the program would enter an endless loop.

```
.
.
.
270 IF ERR=200 THEN RESUME 100
```

If the test condition is met, execution begins at line 80.

RUN

Use RUN to execute a program currently in memory, to start program execution at a specific line, or to load a program from disk into memory and execute it.

When used to load and run a program, RUN clears memory and closes all previously opened data files. If used with the "R" option, data files remain open.

Use the drive option to specify a file located on a drive other than 1.

Syntax

RUN [“[drive:] filename”] [,R]

Examples

RUN “2:list”,R	Loads and executes the file “list” on Drive 2 and leaves all previously opened data files open.
RUN 100	Begins execution of the program currently in memory at line 100.

SAVE*

SAVE stores a file on disk, normally in compressed binary format unless you specify the A option. “A” saves a file in ASCII format, a sometime necessary format that takes more room on disk. For example, the MERGE command requires ASCII format files.

Use the drive option to specify a drive to save a file on other than Drive 1.

The option extended identifier is used to easily identify the type of file saved on disk. Extended identifiers are:

IDENTIFIER	EXPLANATION
.bas	Indicates standard program file.
.asc	Identifies a file saved in ASCII format.
.dat	Specifies data files.

If a file is saved using a file name already on a disk, the file on disk is overwritten.

Syntax

SAVE "[drive:] filename [extended identifier]" [,A]

Example

SAVE "2:test.asc",A	Saves the file "test" on Drive 2 using ASCII format. The extended identifier "asc" identifies the file as being saved with ASCII format.

Note

A file name cannot exceed 6 characters in length, and an extended identifier cannot exceed 3.

SET

SET sets the read-after-write and write-protect attributes to a file or a disk. Set the read-after-write attribute with a capital "R"; set the write protect attribute with a capital "P". Any other character, including small "p" and "r", cancels the current attribute.

Syntax

SET { "filename" / drive },attribute

Examples

SET 1,"R"	Sets Drive 1 to the read-after-write attribute.

SET #1,"P"	Write protects a data file opened as #1.
SET "probl","P"	Write protects the file "probl".
SET 1,"p"	Removes all attributes assigned to Drive 1.
SET 1," "	Removes all attributes assigned to Drive 1.

Note

Attributes set to a disk are held in memory and are only in effect for the current work session. Attributes set to a file are stored on disk and are permanent until changed.

STOP

STOP terminates program execution and returns control to direct mode. You can use STOP at any location in a program. When STOP is encountered, the message "Break in line nnn" is displayed, where nnn is the line number of the statement. To resume execution of the program, specify a CONT statement.

Syntax

STOP

Example

```
10 INPUT A,B,C
20 K=A^2*5.3
30 L=B^2/.26
40 STOP
50 M=C*K+100
60 PRINT M
run
? 1,2,3
PRINT L
15.3846
CONT
115.9
ok
```

SWAP

SWAP exchanges the values of two variables. Swapped variables must be of the same data type or "Type mismatch" is displayed.

Syntax

SWAP variable,variable

Example

```
10 A$="ONE":B$="ALL":C$="FOR"
20 PRINT A$;C$;B$
30 SWAP A$,B$
40 PRINT A$;C$:B$
run
ONE FOR ALL
ALL FOR ONE
ok
```

TERM

TERM puts the PC-8000 in terminal mode. Once in terminal mode, the PC-8000 can communicate with other equipment by the RS-232-C interface. The modifiers of TERM are described as follows.

- word length — establishes the word length for the terminal. An "a" specifies a 7-bit word (used in ASCII); a "j" specifies an 8-bit word (used in JIS).
- parity — is specified by a 0 for no parity, a 1 for odd parity, or a 2 for even parity.
- clock drive ratio — can be 0 (division of 64) or 1 (division of 16).
- auto line feed — a 1 automatically inserts a line feed after each carriage return; a 0 disables auto line feed.

Syntax

TERM word length, parity, clock drive ratio, auto line feed

Example

```
TERM a,0,1,1
```

TRON,TROFF

TRON traces the execution of program statements. Use TRON as an aid in debugging programs. It enables a trace flag that displays each line number of a program as it is executed. The trace flag is disabled by a TROFF, LOAD, or NEW statement.

Syntax

```
TRON
TROFF
```

Example

```
10 K=10
20 FOR J=1 TO 2
30 L=K+10
40 PRINTR J;K;L
50 K=K+10
60 NEXT J
70 END
run
[10] [20] [30] [40]  1   10   20
[50] [60] [30] [40]  2   20   30
[50] [60] [70]
ok
TROFF
ok
```

The numbers enclosed in brackets are program line numbers displayed in the order they are executed.

WAIT

WAIT suspends program execution until a specified machine input port develops a specified bit pattern. The data read at the port is XORed with the integer J, and ANDed with I. If the result is zero, the system will loop back and repeat the process.

Program execution will not resume until the result of the operation is not zero. If J is omitted, it is assumed to be zero.

Syntax

WAIT port, I [,J]

CAUTION

It is possible to enter an infinite loop using the WAIT statement. If this problem occurs, manually restart the machine.

WIDTH

WIDTH determines the number of characters per line and lines per screen for the display. The valid specifications for lines per screen are 20 and 25. Specify characters per line as 80, 72, 40, and 36. The screen default is 40 characters and 20 lines.

Syntax

WIDTH characters [,lines]

Example

WIDTH 80,25	Sets the screen width at 80 characters per line and 25 lines per screen.

NOTE

In color mode, the number of characters and lines displayed is subtracted by one. Hence, a width specified as 80, 20 is actually displayed as 79, 19.

CHAPTER 3

N-BASIC FUNCTIONS

N-Basic functions are described in this chapter. The functions can be called from any program without further definition.

Arguments to functions are always enclosed in parentheses and are abbreviated as follows.

- X and Y — represent numeric expressions.
- I and J — represent integer expressions.
- A$ and B$ — represent string expressions.

If you supply a floating-point value where an integer is required, N-Basic truncates the fractional portion and uses the resulting integer.

ABS

ABS returns the absolute value of the expression X.

Syntax

ABS (X)

Example

```
PRINT ABS (7*(-5))
35
ok
```

ASC

ASC returns a numerical value that is the ASCII code of the first character of the string A$. (See Appendix D for ASCII codes.)

Syntax

ASC (A$)

Example

```
10 A$="TEST"
20 PRINT ASC (A$)
run
84
ok
```

ATN

ATN returns the arctangent of X in radians. The result is in the range $-\pi/2$ to $\pi/2$. The expression X can be any numeric type, but the evaluation of ATN is always performed in single precision.

Syntax

ATN (X)

Example

```
10 INPUT X
20 PRINT ATN (X)
run
?3
1.24905
ok
```

ATTR$

ATTR$ returns the attribute of the drive specified by "drive" or of the file specified by "filenumber" or filename.

Syntax

ATTR$ ({ drive / #filenumber / "[drive:] filename" })

Example

```
PRINT ATTRS ("1:spin")
run
R
```

CDBL

CDBL converts X to a double precision number.

Syntax

CDBL (X)

Example

```
10 X = 454.67
20 Y# = CDBL (X)
30 PRINT Y#
run
454.6700134277344
```

CINT

CINT converts X to an integer by truncating the fractional portion. If X is not in the range −32768 to 32767, an "Overflow" error occurs.

Syntax

CINT (X)

Example

```
PRINT CINT (45.67)
45
ok
```

CHR$

CHR$ returns the ASCII character whose code is I. (ASCII codes are listed in Appendix D.) You normally use this statement to output special characters. For example, execution of the statement CHR$ (7) sounds the buzzer.

Syntax

CHR$ (I)

Example

```
PRINT CHR$ (66)
B
ok
```

COS

COS returns the cosine of X in radians. The calculation of COS (S) is performed in single precision.

Syntax

COS (X)

Example

```
10 X=2*COS (.4)
20 PRINT X
run
1.84212
ok
```

CSNG

CSNG converts X to a single precision number.

Syntax

CSNG (X)

Example

```
10 X#=975.3421#
20 PRINT X#;CSNG (X#)
run
975.3421   975.342
ok
```

CVI, CVS, CVD

These functions convert string values to numeric values. Numeric values that are read from a random disk file must be converted from strings back into numbers. CVI converts a 2-byte string to an integer. CVS converts a 4-byte string to a single precision number. CVD converts an 8-byte string to a double precision number.

Syntax

CVI (2-byte string)
CVS (4-byte string)
CVD (8-byte string)

Example

```
70 FIELD# 1,4 AS N$, 12 AS B$,  . . .
80 GET# 1
90 Y=CVS (N$)
```

DATE$

DATE$ displays the date kept by an internal clock.

The PC-8000 clock keeps track of the time and date. You can use this statement to output the date under control of your program. The date is displayed in the following format:

YY/MM/DD

where YY represents the year, MM represents the month, and DD represents the day. Each item must consist of two digits.

Syntax

DATE$ [="YY/MM/DD"]

Example

DATE$ = "81/01/10" Sets the date to January 10th, 1981.

Note

When power is turned on, the date is automatically set to 79/01/01.

DSKF

DSKF returns the number of clusters unused on the disk specified by drive.

Syntax

DSKF (drive)

DSKI$

DSKI$ assigns the contents of the specified sector to a variable name or to the #0 field buffer.

Syntax

DSKI$ (drive, track, sector)

Example

```
10 A$=DSKI$ (1,19,1)
20 PRINT A$
```

DSKO$

DSKO$ writes the content of random buffer #0 to the specified sector.

The string that you write to the sector must be no longer than 256 characters. If you write a string shorter than 256 characters, the remainder of the sector is filled with zeros.

Syntax

DSKO$ drive, track, sector

Example

DSKO$ 1,5,2

CAUTION

Execution of this statement destroys the previous contents of the sector written to. DSKO$ is an N-Basic statement.

EOF

EOF returns a 1 (true) if the end of a sequential file has been reached. Use EOF to test for end-of-file while inputting to avoid "Input past end" errors.

Syntax

EOF (file number)

Example

```
10 OPEN "list1.dat" FOR INPUT AS#1
20 IF EOF (1) THEN END
30 INPUT #1,N$,S$,D$
40 PRINT N$
50 PRINT S$
60 PRINT D$
70 PRINT
80 GOTO 20
```

ERR, ERL

You use ERR and ERL to handle errors in an error trap routine.

When an error trap is entered, the error code is stored in the variable ERR and the line number where the error occurred is stored in the variable ERL. You normally use ERR and ERL in IF. . .THEN statements to control the flow in the error handling routine.

Error code and line number must appear on the right side of the equal signs. Because ERR and ERL are reserved words, they cannot appear on the right side of the equal sign in LET statements.

To return from an error handling routine, use the RESUME statement.

Syntax

IF ERR = error code THEN. . .
IF ERL = line number THEN. . .

Example

```
10 ON ERROR GOTO 500
.
.
.
```

```
500 REM error handling routine
510 IF ERR = 4 THEN RESUME 100
520 IF ERL = 150 THEN RESUME 150
530 X=15
540 RESUME 0
.
.
.
```

Note

See Appendix C for error code listings.

EXP

EXP returns "e" to the power of X, which must be less than or equal to 87.3366. If EXP overflows, the "Overflow" error message is displayed.

Syntax

EXP (X)

Example

```
10 A = 5
20 PRINT EXP (A-1)
run
54.5982
ok
```

FIX

FIX returns the integer part of X. FIX (X) is equivalent to SGN(X)*INT (ABS(X)). The major difference between FIX and INT is that FIX does not round to the next lower number for a negative value.

Syntax

FIX (X)

Example

```
PRINT FIX (58.75)
58
ok
```

```
PRINT FIX (-58.75)
-58
ok
```

FPOS

FPOS is the same as the LOC function except that the physical sector number is always returned. For details, see LOC.

Syntax

FPOS (file number)

FRE

Arguments to FRE are dummy arguments. If the argument is 0 (numeric), FRE returns the number of bytes in memory not being used by N-Basic. If the argument is a string, FRE returns the number of free bytes in string space.

Syntax

FRE ({ 0 / A$ })

Example

```
PRINT FRE (0)
14542
ok
```

HEX$

HEX$ returns a string that represents the hexadecimal value of the decimal argument. X is truncated to an integer before HEX$ (X) is evaluated.

Syntax

HEX$ (X)

Example

```
10 INPUT X
20 A$=HEX$ (X)
30 PRINT X "DECIMAL IS " A$ " HEXADECIMAL"
run
```

```
?32
32 DECIMAL IS 20 HEXADECIMAL
ok
```

INP

INP returns the byte read from port I. I must be in the range 0 to 255. INP is the complementary function to the OUT statement.

Syntax

INP (I)

Example

```
100 A=INP (255)
```

INPUT$

INPUT$ returns I characters entered from the keyboard or the file opened under the number J. Characters entered from the keyboard are not echoed. All control characters are passed, except for Control-C, which you use to terminate execution of the INPUT$ function.

Syntax

INPUT$ ({ I / I, [#]J })

Example

```
100 PRINT INPUT$ (6)
```

INSTR

INSTR searches for the first occurrence of string B$ in A$ and returns the position where the match is found. Optional offset I sets the position for starting the search. I must be in the range 0 to 255. If I > LEN(A$) or if A$ is null or if B$ cannot be found, INSTR returns 0. If B$ is null, INSTR returns I or 1. A$ and B$ can be string variables, string expressions, or string literals.

Syntax

INSTR ([I,]A$,B$)

Example

```
10 A$ = "ABCDEB"
20 B$ = "B"
30 PRINT INSTR (A$,B$);INSTR (4,A$,B$)
run
 2  6
ok
```

INT

INT returns the largest integer =X.

Syntax

INT (X)

Example

```
PRINT INT (99.89)
99
ok
PRINT INT (-12.11)
-13
ok
```

LEFT$

LEFT$ returns a string composed of the leftmost I characters of A$. I must be in the range 0 to 255. If I is greater than LEN(A$), the entire string (A$) is returned. If I=0, the null string (length zero) is returned.

Syntax

LEFT$ (A$,I)

Example

```
10 A$="DISK BASIC"
20 B$=LEFT$ (A$,4)
30 PRINT B$
run
DISK
ok
```

LEN

LEN returns the number of characters in A$. Nonprinting characters and blanks are counted.

Syntax

LEN (A$)

Example

```
10 A$= "NEC PC-8000"
20 PRINT LEN (A$)
11
ok
```

LOC

With random disk files, LOC returns the next record number to be used if a GET or PUT statement (without record number) is executed. With sequential files, LOC returns the number of sectors (256 byte blocks) read from or written to the file since it was opened.

Syntax

LOC (file number)

Example

```
100 PRINT LOC (1)
```

LOF

LOF returns the largest record number of a random disk file accessed by a PUT or GET statement.

Syntax

LOF (file number)

Example

```
110 IF REC%>LOF (1) THEN PRINT "Amount error"
```

LOG

LOG returns the natural logarithm of X, which must be greater than zero.

Syntax

LOG (X)

Example

```
PRINT LOG (45/7)
1.86075
ok
```

LPOS

LPOS returns the current position of the line printer print head within the line printer buffer. X is a dummy argument.

Syntax

LPOS (X)

MID$

MID$ returns a string of length J characters from A$ beginning with the Ith character. I and J must be in the range 0 to 255. If you omit J or if there are fewer than J characters to the right of the Ith character, all rightmost characters beginning with the Ith character are returned. If I>LEN(A$), MID$ (A$) returns a null string.

Syntax 1

MID$ (A$,I [,J])

Example 1

```
10 A$="GOOD"
20 B$="MORNING EVENING AFTERNOON"
30 PRINT A$;MID$ (B$,8,8)
ok
run
GOOD EVENING
ok
```

Syntax 2

J characters from the Ith character in A$ are replaced by J characters from the top in B$.

```
MID$ (A$,I [,J] )=B$
```

Example 2

```
10 A$="123456789"
20 B$="ABCDEF"
30 MID$ (A$,4,3)=B$
40 PRINT A$
run
123ABC789
ok
```

Also see the LEFT$ and RIGHT$ functions.

MKI$, MKS$, MKD$

These functions convert numeric values to string values. Any numeric value that is placed in a random file buffer with an LSET or RSET statement must be converted to a string. MKI$ converts an integer to a 2-byte string. MKS$ converts a single precision number to a 4-byte string. MKD$ converts a double precision number to an 8-byte string.

Syntax

```
MKI$ (integer expression)
MKS$ (single precision expression)
MKD$ (double precision expression)
```

Example

```
90 AMI=(K+T)
100 FIELD #1, 8 AS D$, 20 AS N$, . . .
110 LSET D$ = MDS$ (AMI)
120 LSET N$=A$
.
.
160 PUT#1
.
.
```

OCT$

OCT$ returns a string that represents the octal value of the decimal argument. X is rounded to an integer before OCT$ (X) is evaluated.

Syntax

OCT$ (X)

Example

```
PRINT OCT$ (24)
30
ok
```

PEEK

PEEK returns the byte (decimal integer in the range 0 to 255) read from memory location I, which must be in range 0 to 65536. PEEK is the complementary function to the POKE statement.

Syntax

PEEK (I)

Example

```
PRINT PEEK (741)
243
ok
```

POS

POS returns the horizontal position of the cursor. (The leftmost position is 0.) The argument is a dummy.

Syntax

POS (I)

Example

```
10 LOCATE 5,5: PRINT POS (I)
run
5
```

CSRLIN

CSRLIN returns the vertical position of the cursor. (The top of the screen is 0.) Unlike the POS function, a dummy argument is not required.

Syntax

CSRLIN

Example

```
LOCATE 4,5:PRINT POS (0); CSRLIN
4   5
```

RIGHT$

RIGHT$ returns the rightmost I characters of string A$. If I≥LEN(A$), then A$ is returned. If I=0, the null string (length zero) is returned.

Syntax

RIGHT$ (A$,I)

Example

```
10 A$="N-BASIC"
20 PRINT RIGHT$ (A$,5)
run
BASIC
ok
```

RND

RND returns a random number between 0 and 1. The value of I will vary the random number returned as follows:

I=0 - generates the same number for a given value I.
I>0 - generates random numbers between 0 and 1.

Syntax

RND (I)

Example

```
10 FOR I=1 TO 5
20 PRINT INT (RND (1)*100);
30 NEXT I
run
24  30  31  51  5
ok
```

SGN

SGN returns the sign of the value X.

Syntax

SGN (X)

If X > 0, SGN (X) returns 1.
If X = 0, SGN (X) returns 0.
If X < 0, SGN (X) returns −1.

Example

```
ON SGN (X)+2 GOTO 100,200,300
```

Control branches to 100 if X is negative, to 200 if X is 0, and to 300 if X is positive.

SIN

SIN returns the sine of X in radians. SIN (X) is calculated in single precision.

Syntax

SIN (X)

Example

```
PRINT SIN (9)
 .412118
```

SPACE$

SPACE$ returns a string of spaces of length X. The expression X is rounded to an integer and must be in the range 0 to 255.

Syntax

SPACE$ (X)

Example

```
10 FOR I=1 TO 5
20 X$ = SPACE$ (I)
30 PRINT X$;I
40 NEXT I
run
 1
   2
     3
       4
         5
ok
```

SPC

SPC prints I blanks and can only be used with PRINT and LPRINT statements. I must be in the range 0 to 255.

Syntax

SPC (I)

Example

```
PRINT "OVER" SPC (15) "THERE"
OVER            THERE
ok
```

SQR

SQR returns the square root of X.

Syntax

SQR (X)

Example

```
10 FOR X = 10 TO 25 STEP 5
20 PRINT X,SQR (X)
30 NEXT X
run
10      3.16228
15      3.87298
20      4.47214
25      5
ok
```

STR$

STR$ returns a string representation of the value of X.

Syntax

STR$ (X)

Example

```
PRINT "$" + STR$ (15+3)
$ 18
ok
```

STRING$

STRING$ returns a string of length I whose characters have ASCII code J or the first character of A$.

Syntax

STRING$ (I,J)
STRING$ (I,A$)

Example

```
10 B$ = STRING$ (10,45)
20 PRINT B$ "MONTHLY REPORT" B$
run
----------MONTHLY REPORT----------

ok
```

Note

In addition, A$ can be specified as any character by enclosing the character in quotes ("-").

TAB

TAB spaces to position I. If the current print position is already beyond space I, TAB has no effect. Space 0 is the leftmost position. I must be in the range 0 to 255. TAB can only be used in PRINT and LPRINT statements.

Syntax

TAB (I)

Example

```
10 PRINT "NAME" TAB (15) "AMOUNT":PRINT
20 READ A$,B$
30 PRINT A$ TAB (15) B$
40 DATA "G.T.JONES", "$25.00"
run
NAME            AMOUNT
G.T. JONES      $25.00
ok
```

TAN

TAN returns the tangent of X in radians. TAN (X) is calculated in single precision. If TAN overflows, the "Overflow" error message is displayed and execution halts.

Syntax

TAN (X)

Example

```
10 G=5
20 Y = G*TAN (6)/2
30 PRINT Y
run
-.727516
ok
```

USR

USR calls your assembly language subroutine with the argument X. "digit" is in the range 0 to 9 and corresponds to the digit supplied with the DEFUSR statement for that routine. See Appendix A.

Syntax

USR digit (X)

Example

```
40 B = T*SIN (Y)
50 C = USR (B/2)
60 D = USR (B/3)
```

TIME$

TIME$ displays or sets the time kept by the internal clock. The time is displayed in the following format:

HH:MM:SS

where the value of HH ranges from 00 to 23 and the values of MM and SS range from 00 to 59. When power is turned on, the time is automatically set 00:00:00. Therefore, the value of TIME$ represents the time period that the machine has been used, unless the time is reset.

Also, you can set the time by entering the following:

TIME$="12:34:56"

Syntax

TIME$ ["time"]

Example

a)
```
PRINT TIME$
12:34:59
ok
```

b)
```
TIME$ = "08:30:40"
```

VAL

VAL returns the numerical value of string A$. If the first character of A$ is not +, −, &, or a digit, VAL (A$)=0.

Syntax

VAL (A$)

Example

```
10 INPUT A$
20 PRINT VAL (A$) +32
run
 25
```

VARPTR

VARPTR is usually used to obtain the address of a variable or array so it can be passed to an assembly language subroutine. A function call of the form VARPTR (A(0)) is usually specified when passing an array, so that the lowest-addressed element of the array is returned.

Syntax 1

VARPTR (variable name)

Returns the address of the first byte of data identified with a variable name. A value must be assigned to a variable name before execution of VARPTR. Otherwise an "illegal function call" error results. Any type variable name can be used (numeric, string, array), and the address returned will be an integer in the range 32767 to −32768. If a negative address is returned, add it to 65536 to obtain the actual address.

Syntax 2

Returns the starting address of the disk I/O buffer assigned to file number.

VARPTR (# file number) [, disk]

Example

```
10 A = 5
20 X = VARPTRA (A)
30 IF X < 0 THEN X=X+65536
40 PRINT X
```

Note

All simple variables should be assigned before calling VARPTR for an array because the addresses of arrays change whenever a new simple variable is assigned.

CHAPTER 4

SEQUENTIAL FILES

A sequential file is a data file that stores its data items one after another in sequence. Sequential files are easy to create but are limited in flexibility and speed when it comes to accessing data. For this reason, sequential files are best suited for jobs where all the data must be read all the time.

The statements and functions that are used with sequential files follow.

OPEN	PRINT#
INPUT#	PRINT# USING
LINE INPUT #	CLOSE
EOF	LOC

NOTE

When the LOC function is evaluated for a sequential file, the returned value is the number of sectors used since the file was opened.

OPEN STATEMENT

Before performing I/O to a sequential file, the file must be opened. The OPEN statement declares the following three items to the system:

a. the name of the file for which I/O is requested,

b. the type of operation being requested (input or output), and

c. the temporary file number.

Example

```
OPEN "data1" FOR INPUT AS #1
```

The example opens the file "data1" for INPUT (reading) and assigns the temporary file number 1. The file number serves as a kind of abbreviated name that is used by the PRINT# and

INPUT# statements to reference the file. Once a file is closed the file number is no longer valid. Hence, each time a file is opened it can be assigned a different number.

File numbers are determined by your response to the system prompt "How many files" displayed when booting up Disk Basic. Your reply determines the number of I/O buffers that will be reserved in memory. The larger the number specified, the less RAM available, thus specify a number no larger than necessary. For most applications, three I/O buffers should be sufficient. A 3 given in reply to the prompt would allow you to use the file numbers 1, 2, and 3.

A sequential file can be opened for INPUT, OUTPUT, or APPEND. INPUT reads a file from disk, OUTPUT writes a file to disk, and APPEND adds data to the end of a previously created file. These modes of operations are indicated by the FOR option in the OPEN command. If FOR is omitted, the OUTPUT mode is assumed.

CREATING SEQUENTIAL FILES

Create sequential files according to the following steps.

OPERATION	EXAMPLE
Open file	OPEN "data1" FOR OUTPUT AS #1
Write data to a file using PRINT# or PRINT USING statement.	PRINT #1,N$;",";S$;",";D$

Example

```
10 REM THIS PROGRAM WRITES A SEQUENTIAL FILE
TO DISK
20 OPEN "data1" FOR OUTPUT AS #1
30 FOR I=1 TO 4
40 INPUT "Name          ";N$
50 INPUT "SECTION       ";S$
60 INPUT "DATE          ";H$
70 PRINT #1,N$;",";S$;",";H$
```

```
80 PRINT
90 NEXT I
100 CLOSE #1
run
NAME?                MICKEY MOUSE
SECTION?             ADVERTISEMENT
DATE?                79/04/01

NAME?                SHERLOCK HOLMES
SECTION?             INVESTIGATION
DATE?                47/09/15

NAME?                AUDREY HEPBURN
SECTION?             PUBLIC RELATIONS
DATE?                52/01/14

NAME?                SUPERMAN
SECTION?             SECURITY
DATE?                79/12/31

ok
```

After a file is created, it probably will have to be read. The read operation requires two steps.

OPERATION	EXAMPLE
Open file	OPEN "data1" FOR INPUT AS #1
Read data from a file using INPUT#or line LINE INPUT# statements.	INPUT #1,N$,S$,D$

Example

```
10 REM PROGRAM TO READ DATA FROM A
SEQUENTIAL FILE
20 OPEN "data1" FOR INPUT AS #1
30 IF EOF (1) THEN END
40 INPUT #1,N$,S$,D$
50 PRINT N$
60 PRINT S$
70 PRINT D$
80 PRINT
90 GOTO 20
```

The EOF statement in line 20 checks for the end of the data file to prevent the program from reading more data items than are on the disk and causing the "Input past end error".

APPENDING DATA TO A FILE

When adding data to a previously created sequential file, open the file in APPEND mode. For example, by simply editing line 20 in the example program to create a file to "OPEN "data1" FOR APPEND AS #1", you can append data to the file "data1". If you tried to add data to "data1" by running the original program, all previously stored data would have been lost.

Example

```
10 REM PROGRAM TO APPEND DATA TO A FILE
20 OPEN "data1" FOR APPEND AS #1
30 FOR I=1 to 4
.
.
.
```

CHAPTER 5

RANDOM FILES

Random files are used when it is necessary to access data from any point within a data file. The random files flexibility is accomplished by structuring the file in units of records that can be accessed individually. A record is a unit of data transferred in one logical I/O operation. To achieve maximum access speed, Disk Basic uses fixed length records with a length of 256 bytes. Data is placed in fields in the records using the FIELD statement.

The statements and functions that are used to create random files follow:

OPEN	FIELD
PUT	CLOSE
MKI$	CVI
MKS$	CVS
MKD$	CVD
LSET	RSET
LOC	LOF
GET	

CREATING RANDOM FILES

The following steps are required for creating random files:

a. Open file

   Example: OPEN "ran" AS #1

b. Execute a FIELD statement to allocate variable space to the random file buffer.

   Example: FIELD #1,20 AS N$,4 AS A$,8 AS P$

c. Execute an LSET or RESET statement to move the data to the random buffer. Numeric data must be converted to strings before it is placed in the buffer. The functions MKI$, MKS$, and MKD$ are used for this operation. You use MKI$ to convert integers to strings, MKS$ to convert single precision numbers to strings, and MKD$ to convert double precision numbers to strings.

   Example: LSET N$=X$
   LSET A$=MKS$ (AMT)
   LSET P$=TEL$

d. Execute a PUT statement to write buffer contents to disk.

   Example: PUT #1, CODE

Example

```
5 REM PROGRAM TO WRITE A RANDOM FILE TO
DISK
10 OPEN "ran" AS #1
20 FIELD #1, 20 AS N$,4 AS A$, 8 AS P$
30 INPUT "Two digit record code ";CODE%
40 INPUT "Name ";X$
50 INPUT "Amount ";AMT
60 INPUT "Telephone ";TEL$
70 PRINT
80 LSET N$=X$
90 LSET A$=MKS$ (AMT)
100 LSET P$=TEL$
110 PUT #1,CODE%
120 GOTO 30
```

The INPUT statement in line 30 initializes CODE% to a record number. The number and the record are both written to disk in line 110. You use the record number to retrieve the record when necessary.

ACCESSING RANDOM FILES

The following steps are necessary to access random files.

a. Open file

   Example: OPEN "ran" AS #1

b. Execute a FIELD statement to allocate random buffer space to input variables.

   Example: FIELD #1, 20 AS N$,4 AS A$,8 AS P$

NOTE

In programs that perform both input and output to the same random buffer, only one OPEN and one FIELD statement can be used.

c. Read the desired record into the random buffer with a GET statement.

   Example: GET #1,CODE%

d. Access the data within the random buffer. Numeric data that was converted to strings can be changed back to numeric using the functions CVI, CVS, and CVD. CVI reconverts integers, CVS reconverts single precision numbers, and CVD reconverts double precision numbers.

   Examples: PRINT N$
   PRINT CVS (A$)

Example

```
10 REM PROGRAM TO READ A RANDOM FILE
20 OPEN "ran" AS #1
30 FIELD #1,20 AS N$,4 AS A$,8 AS P$
40 INPUT "2 digit code ";code%
50 GET #1,CODE%
60 PRINT N$
70 PRINT USING "$$###.##";CVS (A$)
80 PRINT P$
90 PRINT
100 GOTO 30
```

The record, whose number is entered in line 40, is retrieved by the GET statement in line 50 and then displayed.

CHAPTER 6

DISK BACKUP

To minimize any losses that might occur from a damaged disk, backup copies periodically should be made and safely stored away. This chapter describes the "format" and "backup" programs stored on the system disk.

FORMATTING A DISK

Before a disk can be used, it must be formatted. The "format" program is performed by the FORMAT command. The FORMAT command performs level one formatting, which prepares the disk for reading and writing only. No other N-Basic functions, commands, or statements will execute. In this manner, a disk can be used strictly for data files.

Example: FORMAT 2

CAUTION

> Formatting a disk erases all files. Be careful not to destroy any important files.

DISK BACKUP

The program "backup" that comes with the system disk is used to perform level 2 formatting. Level 2 formatting prepares the disk for I/O, copies all files, and allows you to copy the N-Basic operating system. The following is a step-by-step explanation of how to use the backup program.

a. Mount the system disk on drive 1 and load the "backup" program.

b. To copy the system disk, skip to step c. Otherwise, remove the system disk from drive 1 and mount the disk to be copied.

c. Place the destination disk in drive 2.

d. Run the "backup" program. The system displays the prompt:

```
Back up a disk
Mount master disk on drive 1, then hit return.
```

e. Enter a carriage return (press RETURN) in response to this prompt. The system then displays:

```
Mount new disk on drive 2, then hit return.
```

f. Again, press RETURN and the next prompt appears:

```
Format disk 2 (y/n)?
```

g. Enter a "y" if the destination disk has not been previously formatted and the "backup" program formats the disk and begins copying. Enter an "n" if the destination disk has already been formatted and the copying will begin immediately.

Example

```
run "backup"
Back up a disk

Mount master disk on drive 1, then hit return

Mount new disk on drive 2, then hit return

Format disk 2 (y/n)? n
Formatting disk 2
Copying track 0
Copying track 1
Copying track 2
Copying track 3
Copying track 4
Copying track 8
Copying track 9
Complete
ok
```

APPENDIX A

MACHINE LANGUAGE SUBROUTINES

The USR function allows N-Basic programs to call user subroutines, written in machine language, in the same manner as it calls intrinsic functions.

Memory Allocation

Memory space for user subroutines written in machine language must be secured before the program is loaded. As shown in the memory map, the area that can be used is at the very top of user RAM. The size of the area is determined by the second parameter in the CLEAR statement. As the top of user RAM is address 59903 ($E9FF_{16}$), subtract from this value the number of bytes required for machine language subroutines and execute a CLEAR statement with this difference as the second parameter. This action causes N-Basic to consider this value as the upper boundary of its program area, making all locations from that point up to E9FF available for use by machine language routines.

For example, to secure 100 bytes of memory for machine language subroutines, execute the following command.

```
CLEAR 300, 59903-100
```

When machine language subroutines are called by N-Basic, the stack pointer is set so as to allow the subroutine to use up to eight stack levels (16 bytes). If more stack area is required, save the stack pointer, reset it to an area allocated for machine language subroutines, and restore it to its original value before returning to N-Basic.

Calling the USR Function

The format of the USR function follows.

```
USR number (argument)
```

The option number can be an integer from 0 to 9, and (argument) can be any numeric or string expression.

A correspondence is formed between the USR function and a DEFUSR statement for which the same number parameter has been specified, and when the USR function is called program execution jumps to the address specified in this DEFUSR statement.

When the USR function is called, a value is loaded into the A register to indicate the argument type. The values used and their meanings follow:

VALUE IN REGISTER A	ARGUMENT TYPE
2	Two byte integer (2s complement)
3	String
4	Single-precision floating point
8	Double-precision floating point

If the argument type is numeric, the register pair [HL] contains the address of the floating-point accumulator (FAC-3) where the actual value is stored.

When (argument) is an integer:

- FAC-3 contains the lower eight bits of the argument.
- FAC-2 contains the upper eight bits of the argument.

When (argument) is a single-precision floating-point number:

- FAC-3 contains the lower eight bits of the significant value.
- FAC-2 contains the middle eight bits of the significant value.
- FAC-1 contains the upper seven bits of the significant value.

Bit 7 of FAC-1 is the sign bit (0= +, 1 = −).

When (argument) is a double-precision floating-point number, *FAC-7 through FAC-4 contains the additional four bytes of the significant value. (FAC-7 contains the lower eight bits.)

When (argument) is a string, the DE register pair points to the three-byte "string descripter." Byte zero of the string descripter contains the length of the string (0 to 255 bytes), and bytes one and two contain the address of the first byte of the string.

NOTE

When (argument) is a string contained within the program, the DE register pair points at the string. In such a case, there is the possibility that a string within an N-BASIC program will be incorrectly handled or destroyed by the subroutine, causing unpredictable execution results. To avoid this, add the character combination: +" " to a string. For example:

```
A$ = "N-BASIC" + " "
```

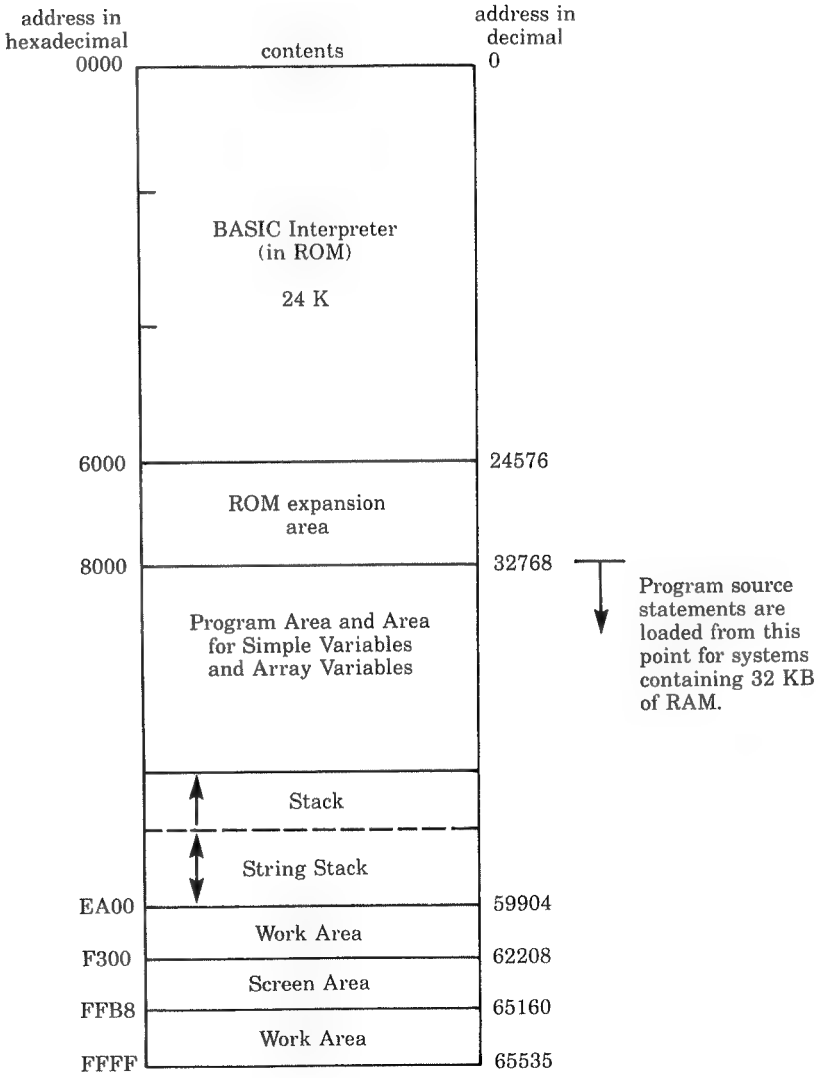
As this string will now be copied to the string space in memory, its copy can be referenced.

Normally, the type of value returned from a user subroutine is the same as the argument specified for the user subroutine. However, an integer value can always be returned in the HL register pair by calling the MAKINT routine. The procedure for calling this routine follows:

```
PUSH H          Save return value
LHLD 24H        Head address of makint
XTHL            Save in stack, restore
                contents of HL
RET             Return
```

APPENDIX B

MEMORY MAP



APPENDIX C

ERROR MESSAGES

Error Code	Error Message	Meaning
1	NEXT without FOR	FOR and NEXT statements do not correspond properly (too many NEXT statements).
2	Syntax error	A line has been encountered that contains an incorrect sequence of characters (such as a misspelled statement).
3	RETURN without GOSUB	A RETURN statement has been encountered for which there is no previous, unmatched GOSUB statement.
4	Out of Data	A READ statement has been executed when there are no DATA statements with unread data remaining in the program.
5	Illegal function call	A parameter that is out of range has been passed to a math or string function.
6	Overflow	The result of calculation is too large to be represented in N-Basic.
7	Out of memory	A program is too large or has too large array variables.
8	Undefined line number	A reference has been made to a line number that does not exist.

Error Code	Error Message	Meaning
9	Subscript out of range	An array element has been referenced with a subscript that is outside the dimension of the array.
10	Redimensioned array	Two DIM statements have been given for the same array.
11	Division by zero	Division by zero has been encountered in an expression.
12	Illegal direct	A statement that is illegal in direct mode has been entered as a direct mode command.
13	Type mismatch	A derived value type does not match that assigned.
14	Out of string space	String variables exceed the amount of allocated string space.
15	String too long	A string is too long (more than 255 characters).
16	String formula too complex	A string expression is too long or too complex. (An illegal nesting level of parentheses, etc.)
17	Can't continue	A nonexecutable CONT command has been encountered. (The pointer is destroyed, etc.)
18	Undefined User Function	A user function has been called before the function definition (DEF statement is given).

Error Code	Error Message	Meaning
19	No RESUME	An error trapping routine with no RESUME statement has been encountered.
20	RESUME without error	A RESUME statement has been encountered before an error trapping routine is entered.
21	Unprintable error	An error message is not available for the error condition that exists.
22	Missing Operand	An expression contains an operator with no operand following it.
23	Line buffer overflow	Line that has too many characters.
24	Position not on Screen	Specified cursor location is out of screen range.
25	Bad File Data	File data on disk is improperly formatted.
26	Disk BASIC Feature	An attempt has been made to execute a disk command with no disk connected.
27	Communication Buffer Overflow	I/O buffer for peripheral devices has overflowed.
28	Port not initialized	The LSI in the port interface is not initialized.
29	Tape read ERROR	An error has been found in input from a cassette tape.

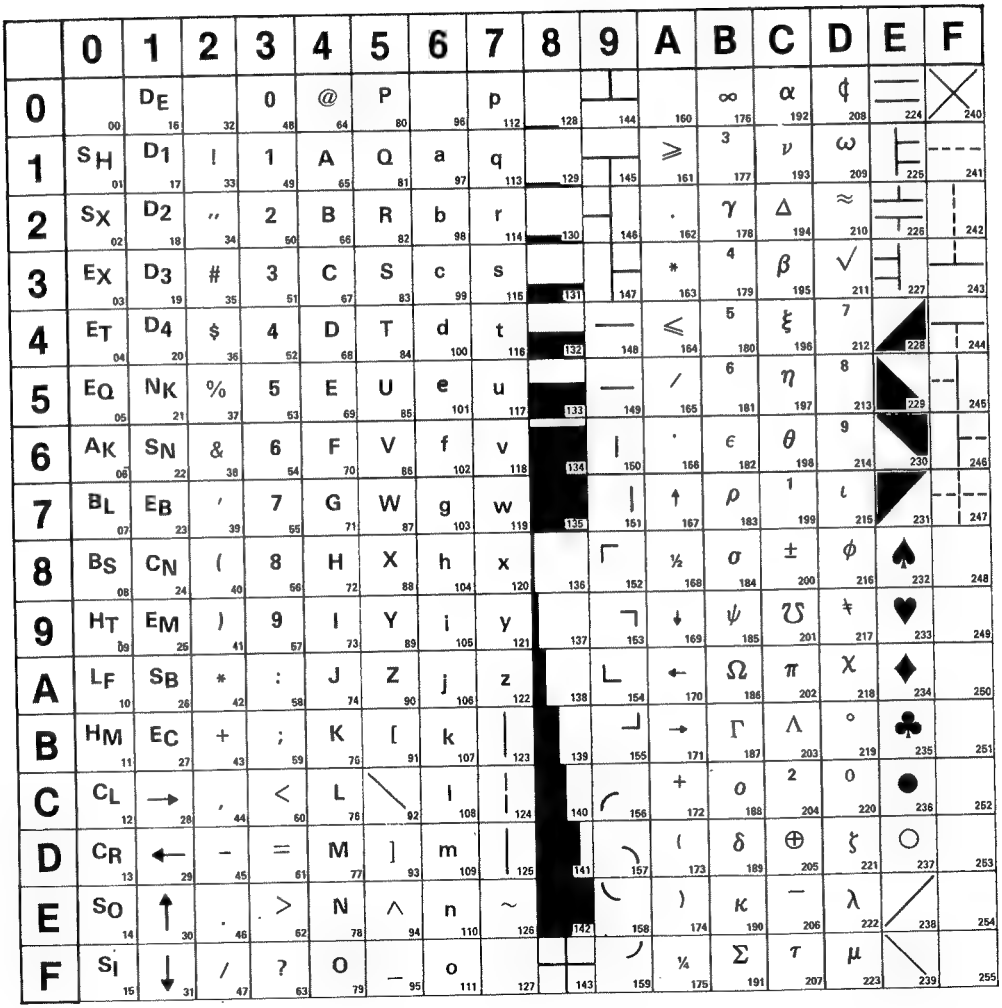
(If a disk unit is not attached, error messages with codes 21 and 26 will be printed.)

50	Field overflow	A FIELD statement is attempting to allocate more than 256 characters of string variables.
51	Internal error	An internal malfunction has occurred in N-BASIC for disk.
52	Bad file number	A statement or command has referenced a file with a file number that is not open.
53	File not found	A LOAD, KILL, or OPEN statement has referenced a file that does not exist on the current file.
54	Bad file mode	An attempt has been made to access a sequential file as a random file or vice versa.
55	File already open	An OPEN or KILL statement has been issued for a file that is already opened.
56	Disk not mounted	An attempt has been made to access a disk that has not been mounted.
57	Disk I/O error	A disk I/O error that cannot be recovered from has occurred.
58	File already exists	The file name specified in a NAME statement is identical to a file name already in use on the disk.
59	Disk already mounted	A MOUNT command was executed on a drive that is already mounted.

Error Code	Error Message	Meaning
60	Disk Full	All disk storage space is in use.
61	Input past end	An INPUT statement has been executed after all the data in the file has been entered.
62	Bad filename	An illegal form has been used for the file name.
63	Direct statement in file	A direct statement has been encountered while loading an ASCII-format file.
64	Bad allocation table	Allocation table has been damaged.
65	Bad drive number	A drive was specified that is not in the system.
66	Bad track/sector	A track/sector number is out of range.
67	Deleted record	Attempt was made to read a deleted record.
68	Rename across disks	A NAME command specified two different drives.
69	Sequential after PUT	A sequential write was attempted after a PUT to a random file.
70	Sequential I/O only	A sequential file was opened as a random file.
71	File not OPEN	An attempt was made to access a file that had not previously been opened.
72	File write protected	An attempt to write to a protected file was made.
73	Disk offline	

APPENDIX D

CHARACTER CODE CHART



Control Characters	Explanation
SH	Start of Header
SX	Start of Text
EX	End of Text
ET	End of Transmission
EQ	Enquiry
AK	ACK (Positive Acknowledgment)
BL	Bell
BS	Back Space
HT	Horizontal Tab
LF	Line Feed
HM	Home (display) on vertical tab (printer)
CL	Clear screen (display) or form feed (printer)
CR	Carriage Return
SO	Shift Out
SI	Shift In
DE	Data Link Escape
D1-D4	Device 1 . . . Device 4
NK	NAK (Negative Acknowledgment)
SN	Sync
EB	End of Transmission Block
CN	Cancel
EM	End of Medium
SB	Substitute Characters
EC	Escape

APPENDIX E

DERIVED FUNCTIONS

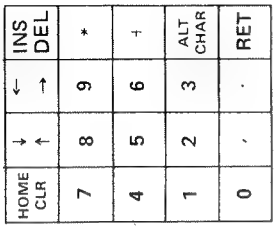
Functions that are not intrinsic to N-Basic can be calculated as follows.

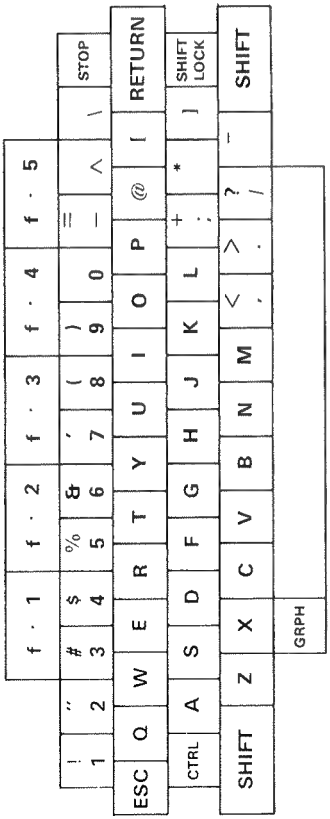
Function	N-Basic Equivalent
SECANT	SEC(X)=1/COS(X)
COSECANT	CSC(X)=1/SIN(X)
COTANGENT	COT(X)=1/TAN(X)
INVERSE SINE	ARCSIN(X)=ATN (X/SQR(−X*X+1))
INVERSE COSINE	ARCCOS(X)=−ATN (X/SQR(−X*X+1)) +1.5708
INVERSE SECANT	ARCSEC(X)=ATN (SQR(X*X−1))+ (SGN(X)−1) *1.5708
INVERSE COSECANT	ARCCSC(X)=ATN (1/SQR(X*X−1))+ (SGN(X)−1) *1.5708
INVERSE COTANGENT	ARCCOT(X)=−ATN(X) +1.5708
HYPERBOLIC SINE	SINH(X)=(EXP(X)− EXP(−X))/2
HYPERBOLIC COSINE	COSH(X)=(EXP(X)+ EXP(−X))/2
HYPERBOLIC TANGENT	TANH(X)=−EXP(−X)/ (EXP(X)+EXP(−X)) *2+1
HYPERBOLIC SECANT	SECH(X)=2/(EXP(X)+ EXP (−X))
HYPERBOLIC COSECANT	CSCH(X)=2/(EXP(X)− EXP(−X))
HYPERBOLIC COTANGENT	COTH(X)=EXP(−X)/ (EXP(X)−EXP(−X)) *2+1
INVERSE HYPERBOLIC SINE	ARCSINH(X)=LOG (X+SQR(X*X+1))

Function	N-Basic Equivalent
INVERSE HYPERBOLIC COSINE	ARCCOSH(X)=LOG (X+SQR(X*X−1))
INVERSE HYPERBOLIC TANGENT	ARCTANH(X)=LOG ((1+X)/(1−X))/2
INVERSE HYPERBOLIC SECANT	ARCSECH(X)=LOG ((SQR(−X*X +1)+1)/X)
INVERSE HYPERBOLIC COSECANT	ARCCHSCH(X)=LOG ((SGN(X)*SQR(X*X +1)+1)/X)
INVERSE HYPERBOLIC COTANGENT	ARCCOTH(X)=LOG ((X+1)/(X−1))/2

APPENDIX F

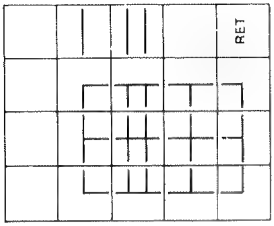
STANDARD PC-8001 KEYBOARD

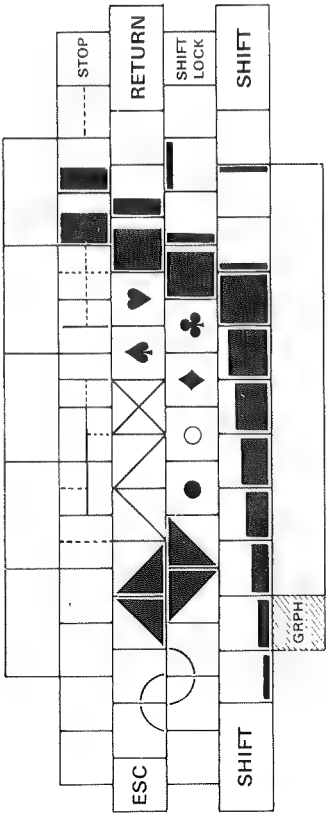




APPENDIX G

GRAPHIC SYMBOL LOCATIONS





APPENDIX H

ALTERNATE CHARACTER SET